National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

UNIVERSITY OF ALBERTA

The Enterprise User Interface and Program Animation Component

by

Greg Lobe

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Science

Department of Computing Science

Edmonton, Alberta
Fall, 1993

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-88183-6

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Gregory L. Lobe

TITLE OF THESIS: The Enterprise User Interface and Program Animation Component

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1993

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly, or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Gregory L. Lobe
6307 84 Avenue
Edmonton, AB
T6B 0H3

September 17, 1993

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled The Enterprise User Interface and Program Animation Component here submitted by Gregory L. Lobe in partial fulfillment of the requirements of the degree of Master of Science.

Duane Szafron

Jonathan Schaeffer

Peter van Beek

Bruce Cockburn

September 16, 1993

Abstract

Enterprise is an integrated graphical environment for developing large-grained distributed programs. It provides a palette of high-level parallelization techniques that allows it to automatically insert code for communication, synchronization, and fault tolerance, freeing the user from tedious, error-prone details. The environment supports designing, coding, compiling, and executing distributed programs. Its program animation component can be used to find certain types of bugs and to tune performance.

This thesis focuses on the user interface and program animation components of the system. It shows that a user interface can be designed and implemented to effectively support the Enterprise programming model, and that animation can be used to tune the performance of distributed systems.

Acknowledgments

I would like to thank my supervisor, Duane Szafron, for his help with the design of the interface and the animation components. As well as creating the basic architecture, he was always available to answer questions or to help when I ran into problems. I would also like to thank him for his patience in waiting for this thesis.

I am grateful to Jonathan Schaeffer for making his experience with parallel programming available. His suggestions about which features were useful, which were not, and which needed to be changed were invaluable. He gave the project a practical grounding.

I would also like to acknowledge the rest of the Enterprise group for their work. Pok Sze Wong and Enoch Chan implemented the first version of Enterprise and their experience led directly to the new version. In addition to Dr. Szafron and Dr. Schaeffer, Ian Parsons, Carol Smith, Paul Lu, Paul Iglinski, Stan Melax, Dr. Z. Yang, and Dr. Greg Wilson all contributed their ideas and criticisms during our weekly project meetings.

And finally, I would like to thank my wife Barbara for her support and love during these past two years.

Table of Contents

Chapter 1. The Enterprise Programming Environment 1

1.1. Introduction 1

1.2. Motivation 1

1.3. The Enterprise Environment 2

1.4. Related Work 4

1.5. Scope and Organization of the Thesis 5

Chapter 2. The Enterprise Programming Model 7

2.1. Module Calls 7

2.2. Module Roles and Assets 8

2.3. Replication and Ordering 10

2.4. Hierarchical Assets 10

2.5. The User Interface 10

2.6. Program Animation 11

Chapter 3. Programming in Enterprise 14

3.1. The Problem 14

3.2. Selecting a Parallelization Technique 15

3.3. Building the Asset Graph 15

3.4. Entering and Editing the Source Code 18

3.5. Setting Compile and Run Options 19

3.6. Compiling Assets and Fixing Syntax Errors 20

3.7. Running the Program 20

3.8. Performance Tuning 22

Chapter 4. The Enterprise User-Interface 26

4.1. Using the Interface 26

4.1.1. Overview 26

4.1.2. The User Model 26

4.1.3. Starting Up 31

4.1.4. The Design Menu 32

4.1.5. Common Operations on Assets 38

4.1.6. Asset menus 41
4.2. Implementation 42
4.2.1. The Control Model 42
4.2.2. Drawing Assets 43
4.2.3. Communicating with Other Components 44
4.2.4. Sub-Directories 45
4.2.5. The Asset Inheritance Hierarchy 47
Chapter 5. Program Animation 49
5.1. The Animation Model 49
5.1.1. The Animation View 49
5.1.2. States 52
5.1.3. Events 53
5.2. Using the Animation System 55
5.2.1. The Animation Menus 56
5.3. Implementation 57
5.3.1. The Animation View 57
5.3.2. The Animation Architecture 58
Chapter 6. Conclusions and Future Work 62
6.1. Unimplemented Features 62
6.2. Problems with the Current Implementation 63
6.3. Future Work 64
6.4. Conclusions 66
References 67
Appendix A. The Enterprise Graph File Format 69
A.1. Notation 69
A.2. Syntax 69
A.3. Semantics 70
Appendix B. The Enterprise Event File Format 72
B.1 Notation 72
B.2 Syntax 72
B.3 Semantics 73

Appendix C. Installation and Setup 75
C.1. Installation 75
C.2. Setup 75
C.3. Other Versions 76

List of Figures

Figure 2.1: A department with 3 component assets 11
Figure 2.2: The animation view of a simple program 12
Figure 3.1: A new Enterprise program 15
Figure 3.2: A program named Animation 16
Figure 3.3: An enterprise containing one individual 16
Figure 3.4: A program containing a line 16
Figure 3.5: An expanded line asset 17
Figure 3.6: The finished asset graph for the Animation program 18
Figure 3.7: The asset options dialog box 19
Figure 3.8: The global options dialog box 19
Figure 3.9: Compiling the sample program 20
Figure 3.10: The Machine File Editor 21
Figure 3.11: Screen display after executing the Animation program 22
Figure 3.12: The animation view of the sample program 23
Figure 3.13: A point in the animation 23
Figure 3.14: The replication dialog box 24
Figure 3.15: Animation view after replicating Split 25
Figure 4.1: Collapsed and expanded enterprise assets 27
Figure 4.2: Collapsed and expanded department assets 28
Figure 4.3: Collapsed and expanded line assets 28
Figure 4.4: Collapsed and expanded division assets 29
Figure 4.5: Dialog box for selecting a file to edit 34
Figure 4.6: Dialog box for global options 36
Figure 4.7: The machine file editor 37
Figure 4.8: The service canvas 37
Figure 4.9: The replication dialog box 39
Figure 4.10: Asset Options Dialog Box 40
Figure 4.11: The asset inheritance graph 47
Figure 4.12: The asset inheritance tree 48

Figure 5.1: A line in the design view 50
Figure 5.2: The line during a replay 50
Figure 5.3: An asset with message queues 51
Figure 5.4: Icons for messages and message queues 51
Figure 5.5: The state transition diagram for enterprise assets 54
Figure 5.5: Message paths 61

List of Tables

Table 4.1: Menu choices for assets 41

Chapter 1.

The Enterprise Programming Environment

1.1. Introduction

This thesis is part of an ongoing research project on the design and implementation of a programming environment for developing distributed programs. The environment, called *Enterprise*, was built with the following objectives in mind:

- to provide a simple high-level mechanism for specifying parallelism that is independent of low-level synchronization and communication protocols,
- to provide transparent access to heterogeneous computers, compilers, languages, networks, and operating systems,
- to support the parallelization of existing programs, and
- to be a complete programming environment that supports programming, compiling, monitoring, and debugging of distributed programs.

Enterprise provides a simple programming model consisting of a small number of commonly used parallelization techniques. The user writes sequential C code, then expresses parallelism by manipulating icons from within a graphical user interface. Enterprise inserts code for communication and synchronization, then compiles and executes the program. The user then uses the system's program animation component for monitoring, debugging, and performance tuning.

The author's contribution to the project involved:

- participation in the design of the programming model [LLM92],
- implementation of the user interface,
- participation in the design of the program animation component [LSS93], and
- implementation of the animation component.

The user interface and animation component are critical to the success of Enterprise, and this thesis describes the design and implementation of these components.

1.2. Motivation

The last decade has seen a steady shift from large central time sharing systems to networks of personal computers and workstations. This is mostly due to advances in hardware technology that have made individual workstations more powerful while simultaneously lowering their cost. Applications that previously required the resources of an expensive mainframe or microcomputer can now be run on relatively inexpensive personal workstations. However, large CPU-intensive applications require more computational power than a single workstation can provide.

A network can be viewed as a large parallel computer, each workstation being an independent processor. The combined processing power available is more than that provided by most supercomputers. Also, many workstations are not used continually, but sit idle on their owners' desks. By taking advantage of these 'idle cycles', users can get

supercomputer performance for their programs, while having a minimal effect on other users of the system.

However, several factors make this difficult:

- Sequential programs must be redesigned as distributed programs. This involves considering parallelization techniques, synchronization, and fault tolerance.
- The programmer must insert low-level communication and synchronization code.
- In a heterogeneous environment, the programmer must consider machine word sizes and the byte and bit ordering within words.
- Different machines may use different operating systems, compilers, or linkers.
- The ordering of events in parallel programs is non-deterministic, which makes them much more difficult to test and debug than sequential programs.
- Finding idle machines involves searching the network and checking machine loads.
- Machine loads may change over time.
- Machines may crash unexpectedly.
- When the program ends, the user must ensure that no processes are left running on other machines.
- Performance tuning requires monitoring several machines and analyzing large amounts of information.

1.3. The Enterprise Environment

Enterprise is a programming environment for designing, coding, debugging, testing, monitoring, profiling, and executing programs in a distributed hardware environment. It has a number of features that distinguish it from other parallel and distributed program development tools:

- Programs are written in a sequential programming language that is augmented by new semantics for procedure calls that allows them to be executed in parallel. Users do not deal with implementation details such as communication and synchronization. Instead, Enterprise inserts all the necessary communication code automatically. Unlike other systems, such as HeNCE [BDG91], Enterprise requires no modifications to existing sequential code.
- Enterprise can generate the code automatically because most large-grained parallel programs make use of a small number of regular techniques, such as pipelines, master/slave processes, and divide and conquer. In Enterprise, the user specifies the desired technique at a high level by manipulating icons using a graphical user interface. The user's code is independent of the parallelization technique selected. The implementation assistants used in the PIE system [LSV89] are similar, but they are not specified graphically and focus on parallel systems with shared memory rather than distributed message passing systems.
- Enterprise uses an analogy between the structure of a parallel program and the structure of an organization. The analogy eliminates inconsistent terminology such

as pipelines, masters, and slaves, replacing it with a consistent terminology based on *assets* such as individuals, departments, and receptionists. Organizations are inherently parallel and often have efficient parallelism. This analogy provides the programmer with a different, but familiar, model for designing parallel programs.

- Enterprise supports the dynamic distribution of work in environments with changing resources. For example, assets can be replicated a variable number of times. During the life of a program, the amount of resources committed to it can vary depending on the resources available.
- Users can control the mapping of processes to processors. Hiding hardware realities of the environment can result in major performance degradation of distributed systems [JS80]. Enterprise is quite flexible in this regard, giving users as much or as little control as they desire. Using a high-level notation, users can specify the processor assignments completely, partially, or leave it entirely up to the environment.
- Enterprise provides global system monitoring to achieve load balancing, detecting when workstations fall idle or become heavily loaded, and monitors system performance for the user.
- Enterprise supports displaying a previous execution of a program with animation of process states and messages. This helps users to find certain types of bugs and identify performance problems. For example, consider an asset that makes a call to another asset, then blocks until it receives a result. If there is a bug in the called asset so that it does not return a result, the user will see the call made, see the caller block, see the called asset receive the call, and see the called asset end without returning anything. The calling asset will remain blocked, and it will be obvious that the problem is that the called asset never returned the result.

Using the graphical user interface, the user draws a diagram of the parallel computation and writes sequential code that is devoid of any parallel constructs. Based on the user's diagram, Enterprise automatically inserts all the necessary code for communication, synchronization, and fault tolerance. It then compiles the routines, assigns processes to processors, and establishes the necessary connections. When the program runs, events like message sends and receives are captured and logged to an event file. The user can then have Enterprise display the execution using the event file. The system displays asset states as they change, displays message queues, and animates messages as they move between assets. The user can pause, single step, or restart the animation at any time, expand message queues to examine the messages they contain, or expand messages to view their parameters.

Enterprise's user interface was designed to allow a user to express parallelism in a simple graphical manner. Although other parallel programming environments like HeNCE support graphical views, these views are either non-editable or are edited by drawing nodes and arcs that represent processes and communication paths. In Enterprise, the application graph is an asset graph and it is constructed in a novel way. The user starts with an enterprise asset and constructs the graph by re-classifying, expanding, and

replicating individual icons. This approach has several advantages over an arbitrary graph structure:

- Enterprise assets represent high-level parallelization techniques, not individual processes. This allows the user to design at a higher level of abstraction.
- Assets themselves are not drawn and connected by the user in an arbitrary manner. Instead, assets are re-classified and expanded to create a program. This reduces the drawing errors that result from indiscriminately connecting and disconnecting nodes using arcs.
- Using re-classification to create programs prevents cycles of asset calls and thus prevents the possibility of deadlock.
- The structure of an Enterprise program clearly indicates the type and degree of parallelism. The flow of information is reflected in the call structure, the type of parallelism is reflected in the asset type, and the degree of parallelism is reflected in the asset replication factors.
- Enterprise manages program complexity by allowing assets to be expanded and collapsed so that the program can be viewed at different levels of abstraction.
- Experimentation is encouraged because the parallelization technique is specified graphically and is independent of the code.

The program animation component was designed to help users debug, test, and monitor their programs. A user can watch a program run graphically. There is no need to try to infer what is happening during a run by techniques such as monitoring machine loads or inserting special code. The entire program is shown on the screen at once. Events that occur concurrently actually happen concurrently on the display, greatly simplifying the task of determining exactly what a program is doing when there are bugs or performance problems. It uses graphics to present large amounts of data in a highly organized, easy to understand form. The program can be stepped to any point, stopped, and messages with their parameters can be examined. Bottlenecks can be identified by watching for messages building up in message queues. Wasted resources can be identified by watching for idle assets.

1.4. Related Work

Some similar work developing integrated environments for distributed programming has been done, but few make use of program animation.

The JADE system described in [JLU87] has parts that are similar to Enterprise. It is a complete environment that includes a distributed kernel and window system. Unlike Enterprise, however, users do not write normal sequential code, but must write programs using JADE's primitives to express the parallelism in the code explicitly rather than graphically. For animation, data collection is separated from data display by using *sensors* attached to processes to collect data and *consoles* to display the data captured by the sensors. Several consoles have been implemented, including a graphical one called a *mona* console that is similar to the Enterprise animation display. Processes are represented by icons and the state of each process is indicated graphically. Unlike Enterprise, JADE

allows the user to position the icons. Messages do not move on the display, but the icons indicate when messages are sent and received.

David Taylor has implemented a debugger for distributed programs that use the Hermes system [Tay92]. Like the Enterprise animation system, Taylor's debugger is event-based and uses animation of events and messages to help the user understand the interaction between processes. Taylor presents the logical partial ordering of events rather that an imposed total ordering [Fid88] to avoid misleading the user with fictitious information, but he makes no attempt to indicate the elapsed time between events. This contrasts with Enterprise, which attempts to present events in the order in which they occurred in real time, and to model the real elapsed time between them. Taylor's event display uses a time-process diagram where events are plotted along one axis, and time is plotted along the other. Enterprise animation displays a sequence of snapshots of the state of the entire program.

The PIE system described in [SR85] and [LSV89] has some similarities to Enterprise, but it focuses on shared-memory parallel systems rather that large-grained distributed systems. PIE provides an integrated programming environment that includes a program animation component. As in Enterprise, separate views are used for programming and for animation. PIE programs use a language-independent meta-language to express parallelism by building a hierarchical structure of communicating processes. Unlike Enterprise however, PIE provides real-time animation. The user marks what to monitor, and the system uses sensors to collect data at run-time. A variety of animation displays are supported, including a time-process graph of execution times and an animated process invocation tree. PIE uses objects called *implementation assistants* to express parallelism at a high level, much like Enterprise does with assets. An implementation assistant is a template for the decomposition of a parallel computation and the way it is controlled.

The system most similar to Enterprise is HeNCE [BDG91]. Like Enterprise, it is a complete environment for developing distributed programs. The user expresses parallelism graphically, independently from the code. The system then inserts code for communication and synchronization, captures events at run time, and can replay the execution using animation. The parallelism graph is built by drawing arcs representing dependencies and control flow between nodes representing subroutines. Unlike Enterprise, HeNCE allows users to draw arbitrary graph structures. This allows the possibility of cyclic graphs which may cause deadlock or livelock at run time. In addition, HeNCE requires the parameters for calls to be specified in the graph as well as in the code for the call. In some ways, the system is more sophisticated than Enterprise. For example, it supports heterogeneity by allowing different implementations of nodes for different architectures, and uses checkpointing to achieve fault-tolerance.

1.5. Scope and Organization of the Thesis

The emphasis of this thesis is on the Enterprise user interface and its program animation component. Both were implemented as a result of the thesis research. The thesis is organized as follows:

- Chapter 2 describes the Enterprise model and gives an overview of the user interface and program animation components.

- Chapter 3 gives an example of writing, compiling, running, animating, and tuning a program from the user's point of view.
- Chapter 4 gives a detailed discussion of the user interface and how it was implemented.
- Chapter 5 gives a detailed discussion of the animation system and how it was implemented.
- Chapter 6 summarizes the thesis and discusses the ongoing research.

Chapter 2.

The Enterprise Programming Model

The overall organization of a parallel or distributed Enterprise program is similar to the organization of a sequential program. The structure of an application program is unaffected by whether it is intended for sequential or distributed execution. The user views an Enterprise program as a collection of modules. Each module consists of a single entry procedure that can be called by other modules and a collection of internal procedures that can be called only by other procedures in that module. No common variables among modules are allowed. In many ways, this is analogous to programming with abstract data types, which provide well-defined means for manipulating data structures while hiding all the underlying implementation details from the user.

Within any module, the code is executed sequentially. For example, a sequential program consists of a single module whose entry procedure is the main program. Enterprise introduces parallelism by allowing the user to specify the way in which the modules interact. Module interaction is specified by two factors: the role of a module and the invoking call to a module. The role of a module defines which one of a fixed set of parallelization techniques, or asset kinds, the module will use when it is invoked. The call to a module defines the identity of the called module, the information passed, and the information returned. The role of a module is specified graphically whereas the call is specified in the code.

2.1. Module Calls

In a sequential program, procedures communicate using procedure calls. The calling procedure, say `A`, contains a procedure call to a procedure, say `B`, that includes a list of arguments. When the call is made, procedure `A` is suspended and procedure `B` is activated. Procedure `B` can make use of the information passed as arguments. When procedure `B` has finished execution, it can communicate results to procedure `A` via side-effects to the arguments and/or by returning a value if the procedure is in fact a function.

Enterprise module calls are similar to sequential procedure calls. As with procedure and function calls, it is useful to differentiate between module calls that return a result and those that do not. Module calls that return a result are called *f-calls* (function calls) and module calls that do not return a result are called *p-calls* (procedure calls). Conceptually, there is no difference between a sequential function call and an Enterprise module call except for the parallelism.

An Enterprise f-call is not necessarily blocking. Instead, the caller blocks only if the result is needed and the called module has not yet returned. Consider the following example:

```
result = B( data );
/* some other code */
value = result + 1;
```

When this code is executed, the arguments of `B (data)` are packaged into a message and sent to `B`. `A` continues executing in parallel with `B`. However, when the calling module tries to access the result of the call to `B (value = result + 1)`, it blocks until `B` has returned a message containing its result.

There is no syntactic difference between procedure calls and module calls. This makes it easier to transform sequential programs to parallel ones and makes it simple to change parallelization techniques using the graphical user interface, without making changes to the code.

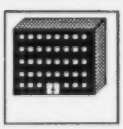
2.2. Module Roles and Assets

The role of a module is based solely on a parallelization technique and is independent of its call. There are a fixed number of pre-defined roles corresponding to asset kinds.

Enterprise uses an analogy between distributed programs and the structure of an organization to help describe module roles. In general, an organization has various assets available to perform its tasks. For example, a large task could be divided into sub-tasks where various sub-tasks are given to different parts of the organization such as individuals, departments, assembly lines, or divisions to perform in parallel. In addition, an organization usually provides such standard services as time keeping or information storage and retrieval.

Currently, Enterprise supports the roles corresponding to six different asset kinds: enterprise, individual, department, line, division and service. In addition, two other specialized individual assets are defined: receptionist and representative.

Enterprise



An *enterprise* is a single program. It is analogous to one organization or enterprise.

Individual

An *individual* contains no other assets. An individual is analogous to an individual person in an organization. For example, a clerk in a grocery store is an individual. When called, an individual executes its sequential code to completion. Therefore, any subsequent call to the same individual must wait until the previous call is finished. The individual can be viewed as a process executing a sequential program. In general, an individual can be re-classified as another asset kind at any time. However, there are two special kinds of individuals: receptionists and representatives. *Receptionists* serve as the first element of a department, line, or division. They provide the name by which the asset can be called and serve as its entry point. Receptionists can't be re-classified as any other asset type. *Representatives* are used in divisions as the leaves of a parallel recursion tree. They can be re-classified as divisions only. Each individual, including receptionists and representatives, has code associated with it.

Line

A *line* contains a fixed number of heterogeneous assets in a fixed order. Each asset contains a call to the next asset in the line. This is analogous to a construction, manufacturing, or assembly line in an organization where at each point in the line, the work of the previous asset is refined. For example, a line might consist of an individual who takes an order, a department that fills it, and an individual that addresses the package and mails it. A subsequent call to the line waits only until the first asset has finished its sub-task for the previous call, not until the entire line is finished. The first asset in a line is a receptionist that shares its name with the line. It is the only asset that is externally visible. That is, the first asset of a line is the only asset that may be called from another asset. Lines are often referred to as pipelines in the parallel computing literature.

Department

A *department* contains a fixed number of heterogeneous assets. A single receptionist asset shares its name with the department so that it can be called by other assets. The other assets in a department are called directly by the receptionist and are not allowed to call each other. A department is analogous to a department in an organization where a receptionist is responsible for directing all incoming communications to the appropriate place. Note that in our analogy, a department consists of a collection of assets of any kind: individuals, departments, lines and divisions. A department has no analogous term in the parallel computing literature.

Division

A *division* contains a hierarchical collection of identical assets where work is divided and distributed at each level. Divisions can be used to parallelize divide-and-conquer computations. When a division is created, it has a single receptionist asset that shares its name with the division so that it can be called by other assets. In addition, it has a single representative asset that represents the parallel recursive call made by the receptionist to the division itself. When the receptionist makes a recursive call, it is calling the representative. When a representative makes a recursive call, it is making a normal sequential recursive call to itself. The user may change the breadth of the division's first level by replicating the representative. The user may add a level to the depth of the recursion by re-classifying the representative as a division. The new division contains a receptionist and a representative. The breadth of the tree at any level is determined by the replication factor of the representative or division it contains. This approach is capable of specifying arbitrary fan-out at each level of the division. Divisions are the only recursive assets in Enterprise.

Service

A *service* contains no other assets. Services cannot be replicated, but they can be called by any other asset in the program. Work is processed on a 'first-come / first-served' basis. A service is analogous to any asset in an organization that is not consumed by use and whose order of use is not significant. For example, a clock on the wall and a counter that records the total number of vehicles that have passed through several toll booth lanes can be considered services.

2.3. Replication and Ordering

As well as the module role, there is another dimension to parallelism in Enterprise: replication. When an asset is *replicated*, more than one process can simultaneously execute its code. The user specifies a minimum and maximum replication factor, and Enterprise dynamically allocates as many processors as are available, up to the maximum. Each processor will run a copy of the code for the asset. When a replicated asset is called, the first available replica does the work. If all replicas are busy, the call waits for the first one that becomes free.

Replication and asset type are orthogonal — all assets except receptionists can be explicitly replicated. A receptionist cannot be explicitly replicated since it serves as the entry point for another asset, but it is implicitly replicated when the asset that contains it is replicated.

Replicated assets may be ordered or unordered. If an asset is unordered, any reference to the return value of the asset will receive the first value returned from any replica of that asset. If an asset is ordered, the first reference to the return value of an asset will receive the return value of the first call. Assets are ordered by default.

2.4. Hierarchical Assets

Enterprise assets can be built from any combination of assets. For example, a user could construct a department where one subordinate asset is itself a line of individuals and another is a division. The model allows the user to re-classify an asset without any changes to the source code. The only change that might occur is the gathering or separation of functions from one file to another.

Lines, departments, divisions, and enterprises are all called *composite assets* because they contain other assets. The assets they contain are called their *components*. If an asset is a component of a composite asset, the composite asset is called its *parent*.

2.5. The User Interface

The Enterprise user interface provides a complete environment for writing, compiling, executing, and testing distributed programs. Source code for assets can be entered using a standard text editor, parallelism can be specified graphically by manipulating icons that represent assets, and the compiler can be invoked for any part of the asset hierarchy with error messages displayed in a window for use while making corrections. The program can then be executed with the output of each asset going to its own separate window, and animation can be used to replay the program. All of this can be done without leaving the environment.

Assets are displayed as icons. Actions are performed by making selections from context sensitive menus linked to the assets. Pressing the middle mouse button causes a menu to appear at the mouse cursor location. If the cursor is over an asset, the menu for that asset is displayed. The menus for assets contain only those actions that can be applied to the asset in its current context. For example, the enterprise asset cannot be deleted, so there is no Delete action in its menu. When the cursor is not over any asset, the menu for global program operations is displayed.

The main Enterprise window consists of one or two canvasses. The *program canvas* is used to display and edit the graphical representation of the program. The

service canvas contains the services that the program can use. It can be hidden or displayed. Both canvasses use the same menu of global program operations. The menu contains choices like Save Program and Compile Program that allow the user to perform operations on the program as a whole.

When Enterprise is first started, the program canvas contains a single enterprise asset that itself contains a single individual asset. The user builds the program by modifying this individual asset. For example, to build the department of three assets shown in Figure 2.1 the user would do the following:

- create a new enterprise named ABC,
- expand the enterprise asset to reveal the individual it contains,
- re-classify the individual as a department,
- expand the department, which will contain a receptionist and a single individual by default,
- add one more individual to the department,
- name the assets A, B, and C, and
- replicate the second component of the department.

The source code for the three components can then be entered and the program can be compiled and executed. Chapter 4 gives a detailed description of the interface and explains the elements visible in Figure 2.1. Chapter 3 contains a complete example of building an Enterprise program.

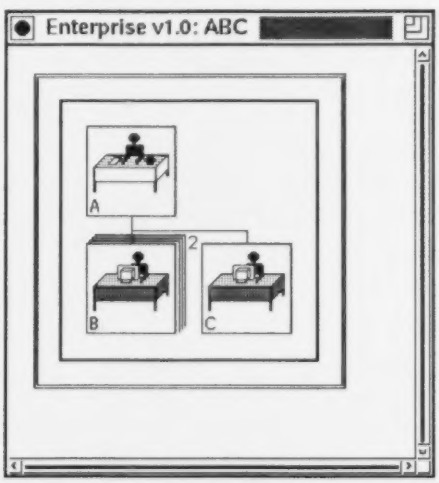


Figure 2.1: A department with 3 component assets

2.6. Program Animation

Enterprise also provides a program animation facility that can be used to monitor a program's performance and to identify parallel programming and logic errors. The system captures events as the program runs and logs them to an event file. Animation then displays the execution using the event file.

When a program runs, assets change state as they send messages and replies to each other. The animation system displays the state of each asset, animates messages and

replies as they move between assets, and displays message queues. There are no facilities for setting breakpoints or examining the values of variables.

Animation is displayed using a different view of the program than the one used for editing. When Animate is selected from the program canvas menu, the *design view*, described above, is replaced with the *animation view*. In this view, each asset that is replicated is expanded to show all of its replicas. In addition, assets have message queues for incoming calls and received replies, and the menus contain actions for animation instead of editing.

The menus for assets contain choices for expanding or collapsing the assets. The menu for the animation view itself contains choices to pause or restart the animation, single step, set animation options, or end the animation and switch back to the design view. Figure 2.2 shows the animation view of the department in Figure 2.1 at a point during a replay.

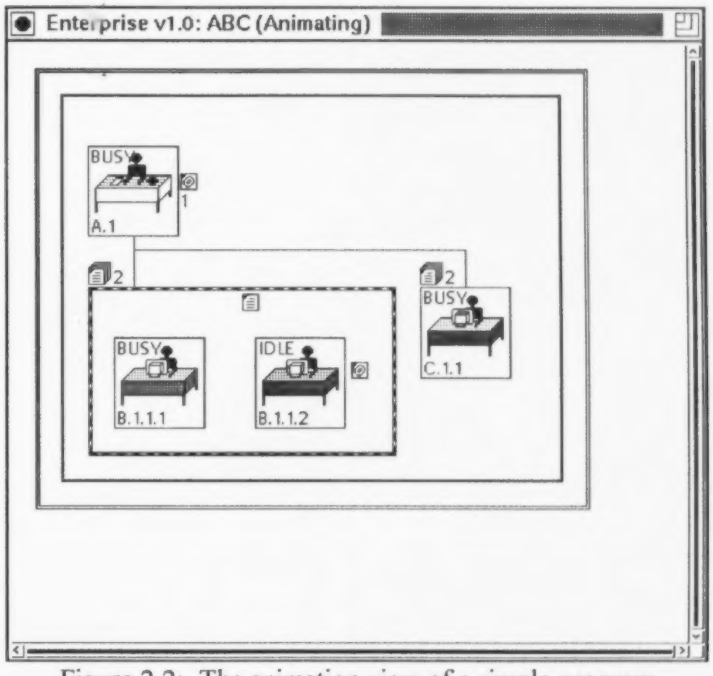


Figure 2.2: The animation view of a simple program

At this point in time, A has sent work to B and C, and is now busy doing some other work. It has a result from a previous call waiting in its reply queue to its right. C is busy working on a job and has two more waiting in its input queue at its top left. B has two replicas that share a common input queue. There are two jobs waiting in the queue. B.1.1.1 is currently working on a job. B.1.1.2 has just completed a job and the result can be seen on its right hand side on its way back to A. At the same time, a new job is on its way from the shared input queue to B.1.1.2.

Chapter 5 describes the animation system. The example in Chapter 3 includes program animation.

Chapter 3.

Programming in Enterprise

This chapter contains a complete example of building a distributed program using Enterprise. Parallelizing an application using Enterprise involves the following main steps:

- Divide the problem into assets and select one of Enterprise's parallelization techniques for each asset.
- Build the asset graph.
- Enter the source code for the assets.
- Set compile and run options.
- Compile the assets and fix any syntax errors.
- Run and debug the program, fixing any logic errors
- Tune the program's performance.

3.1. The Problem

Consider a program called *Animation* that displays a group of animated fish swimming across a display screen. This problem was contributed by the graphics research group in our department. The program computes frames for the animation using three major procedures: *Model*, *PolyConv*, and *Split*. They have the following functionality and pseudo-code:

- The main procedure, Model, computes the location and motion of each object in a frame, stores the results in a file, calls PolyConv to process the frame, and goes on to the next frame:

```
Model()
{
        for each frame
        {
                compute location of objects, generate frame
                PolyConv( frame );
        }
}
```

- PolyConv reads a frame from a disk file, then performs data format transformations, viewing transformations, projections, polygon sorting, and back-face removal. When these are done it calls Split:

```
PolyConv( frame )
{
        perform transformations and projections
        Split( polygons, frame );
}
```

- Split performs hidden surface removal and anti-aliasing, then stores the final rendered image in another file:

```
Split( polygons, frame )
{
      perform hidden surface removal and anti-aliasing
}
```

3.2. Selecting a Parallelization Technique

Examining the structure of the program shows that Model consists of a loop that, for each frame in the animation, performs some work on the frame and calls PolyConv with the results. PolyConv manipulates the frame produced by Model and calls Split. Split does the final polishing of the frame and writes the final image to disk.

There is no reason why Model should wait for PolyConv to finish a frame before starting the next one. Model could be working on the second frame while PolyConv is working on the first. Similarly, PolyConv does not need to wait for Split. This type of parallelism is called a pipeline in the literature. In Enterprise it can be represented by a line asset. Model, PolyConv, and Split will be three components of the line. The line will share its name with its first component, so it will be named Model. That is, we will build a line named Model that has three components: a receptionist named Model, an individual named PolyConv, and an individual named Split.

3.3. Building the Asset Graph

Building the asset graph is done from the Enterprise user interface. When Enterprise is installed, it sets up a directory structure in a sub-directory specified by the user. To run Enterprise the user must make that directory the current directory and enter the command `'st80'`, which starts a Smalltalk interpreter. Once Smalltalk starts, the user must evaluate the expression `'Enterprise open'`. This can be done by typing it into a workspace, selecting it, and then selecting doit from the middle mouse button menu. The user interface will then appear as shown in Figure 3.1.

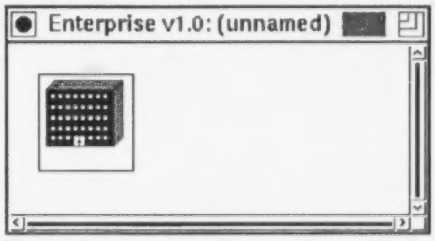


Figure 3.1: A new Enterprise program

As mentioned in Chapter 2, the program canvas initially contains a new enterprise asset. To build the graph we modify this enterprise asset. The first step is to name the program using the context-sensitive asset menu. Moving the cursor so it is over the enterprise asset and pressing the middle mouse button will display the menu, which contains the single choice Name. Moving the cursor to Name and releasing the button will display a dialog box. Typing the name *Animation* into the dialog box and pressing ENTER names the program, with the result shown in Figure 3.2.

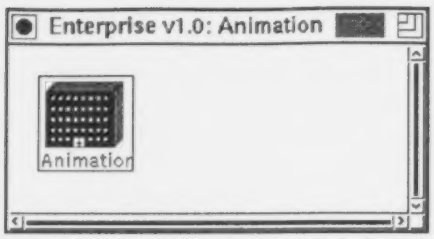


Figure 3.2: A program named Animation

The asset menu now changes to contain the choices Name and Expand. Selecting Expand from the menu will expand the enterprise asset to reveal the single individual it contains, giving the view shown in Figure 3.3. Note that the individual has been given a default name of unnamed1.

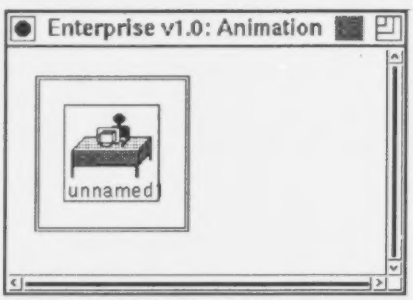


Figure 3.3: An enterprise containing one individual

Selecting Line from the asset menu for the individual will change the individual to a line. Naming the line *Model* by selecting Name from the line's asset menu gives the view shown in Figure 3.4. The icon now represents a line, and the numeral 1 indicates that the line currently consists of a receptionist and one individual.

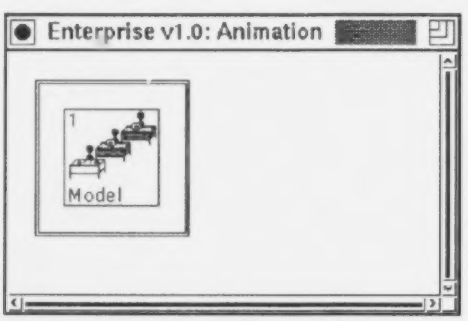


Figure 3.4: A program containing a line

The line can be expanded by selecting Expand from its asset menu. Doing this reveals a receptionist named Model and an unnamed individual as shown in Figure 3.5. The double line represents the enterprise, the dashed line represents the line, and the icons represent the receptionist and individual. Clicking inside the enterprise rectangle but outside the line rectangle will display the asset menu for the enterprise asset. Clicking inside the line rectangle but outside either of its component's icons will display the asset

menu for the line. Clicking on the icons for the receptionist or individual will display their respective asset menus.

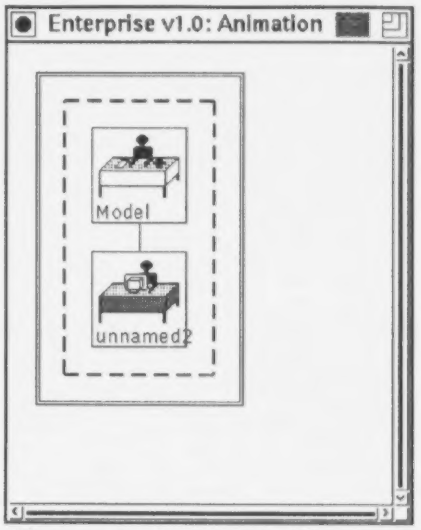


Figure 3.5: An expanded line asset

The second component of the line can now be named PolyConv by choosing Name from its asset menu. A third component can then be added by selecting Add After from PolyConv's menu. The new individual can then be named Split using its own asset menu. The finished graph is shown in Figure 3.6.

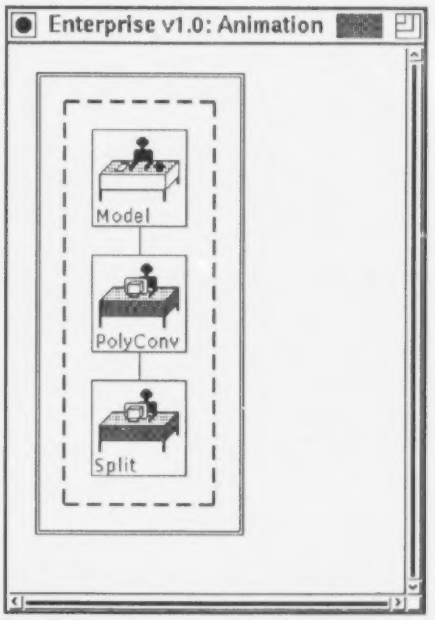


Figure 3.6: The finished asset graph for the Animation program

3.4. Entering and Editing the Source Code

Enterprise includes an integrated editor for editing asset source code. The editor can be one selected by the user, or it can be the one provided by Enterprise. When the system starts up, it reads a file named .ENTrc in the user's home directory. This file contains global system options, one of which is the editor to use. To specify an editor, the user inserts the line `'EDITOR=<editor_cmd>'` where `<editor_cmd>` is the command that invokes the editor.

To invoke the editor, the user selects Code from the asset menu for an asset. If source code already exists for the asset, the file is read into the editor, otherwise a new file is created. There can be several editors open at the same time, one for each asset. If the built-in editor is used, text can be copied and pasted between assets.

The code for an asset consists of an entry function, with the name of the asset, plus any support functions it calls. Procedures that are common to several assets can either be placed in a separate file that will be automatically compiled and linked with the asset code, or the user can build a library and have Enterprise link it with the asset modules.

For this example, existing code needs to be organized into files for Model, PolyConv, and Split. For each of the assets, we select Code from its menu to display an editor. Then we read in the original sequential code and distribute it to the appropriate assets. Note that there is no code associated with the line as a whole, so there is no Code action in the line asset's menu.

3.5. Setting Compile and Run Options

Before compiling and running the program, the user can set some options. Selecting Options from an asset's menu displays a dialog box for setting compile and run options for that asset. Selecting Options from the design view menu brings up a dialog box for setting global compile and run options. The boxes are shown in Figures 3.7 and 3.8. They are described in more detail in Chapter 4.

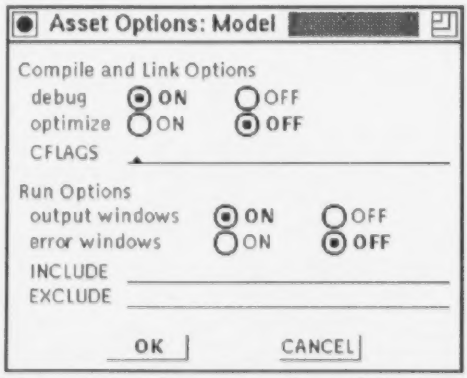


Figure 3.7: The asset options dialog box

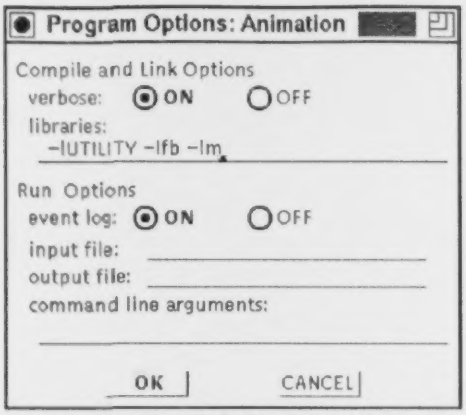


Figure 3.8: The global options dialog box

For our example, because we want to use the same set of options for Model, PolyConv, and Split, we set them in the dialog box for the line asset Model. Setting them for the line sets them for all the components of the line. The line's menu is invoked by clicking inside the dotted line rectangle but outside the asset icons shown in Figure 3.6. We set the options as shown in Figure 3.7. The global program options are set as show in Figure 3.8.

3.6. Compiling Assets and Fixing Syntax Errors

The next step is to compile the code and fix any syntax errors. When the assets are compiled, the Enterprise precompiler inserts the parallelization code, then invokes a standard C compiler and linker to produce the executable program. The compiler can be started from either the design view menu or from an asset menu. If it is started from an asset menu, only the asset is compiled and the linker is not run. If the asset is a line, department, or division, then all of its components are also compiled. If the compiler is started from the design view menu, the entire program is compiled and linked. The system contains a built-in "make" facility, so only those assets that have changed since the last compilation will actually be compiled. Only one compilation can be active at a time.

When the compiler is invoked, a window is opened that will contain messages from the compiler, including error messages. The output at a point during the compile of the sample program is shown in Figure 3.9. Note that the verbose option was enabled for this example.

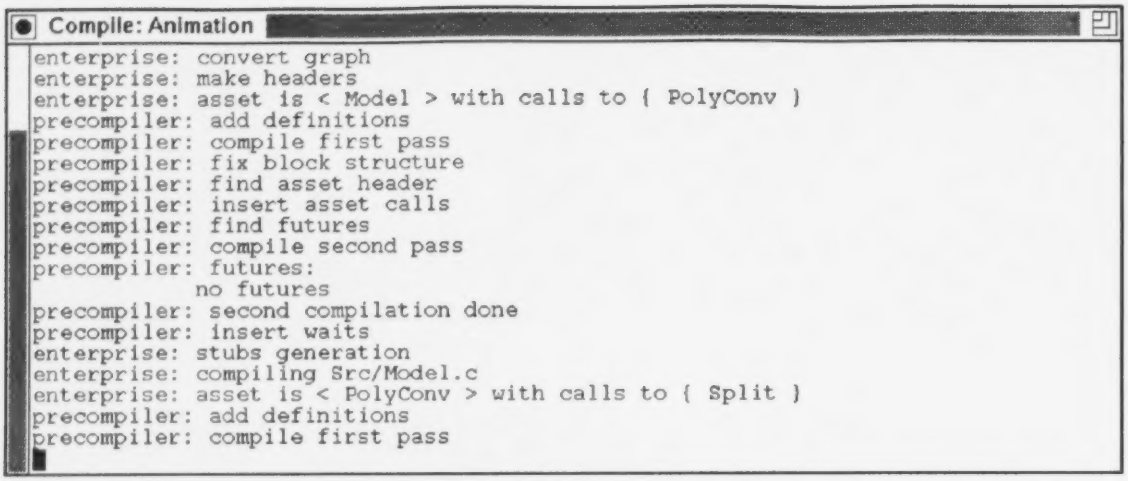


Figure 3.9: Compiling the sample program

If there are errors, the user can leave the compiler window open, invoke editors on the assets, and fix the errors. When the program is being re-compiled, it will use the same compiler window. If there are no errors, the window can be closed to conserve screen space.

3.7. Running the Program

Before the compiled program can be executed, the system must be told which machines it can use. This is done by selecting Edit Machine File from the design view menu. The machine file editor will then open as shown in Figure 3.10. Here we specify a set of machines that Enterprise will use to execute assets. The current set of machines are displayed in a list box. Typing the name of a machine in the text field below the box, then clicking the Add button will add the machine name to the list. Selecting a machine name from the list, then clicking the Delete button will delete the machine from the list. Clicking the OK button saves the list and exits the editor.

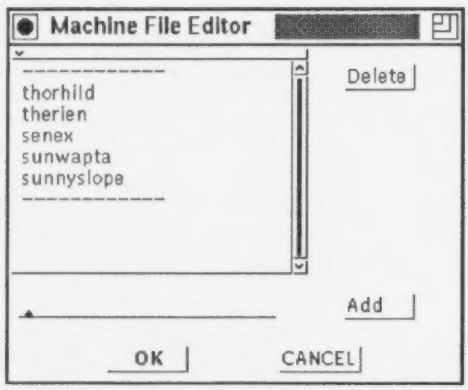


Figure 3.10: The Machine File Editor

Once the machines have been specified, the program can be executed. This is done by selecting Execute from the design view menu. Note that when we set the options in Section 3.5, we elected to have output windows opened for each asset. This means that when the program is run, a window will be created for each asset that will contain any output the asset writes to its standard output file. After the run, the windows are left open until explicitly closed by the user.

A console window is also opened for the run. This window will contain messages from the Enterprise run-time executive. It is left open after the run for use as a shell by the user for examining the results of the program. The window closes when the shell is exited.

The program runs in a sub-directory of the directory where Enterprise resides. Each program has its own directory whose name is based on the name of the program, as described in Chapter 4. The program runs from this directory. The sample program reads two files from its current directory and requires that a sub-directory named data exists. After the files were copied and the data sub-directory was created from a UNIX shell running outside the interface, the program was executed. Figure 3.11 shows the resulting screen display.

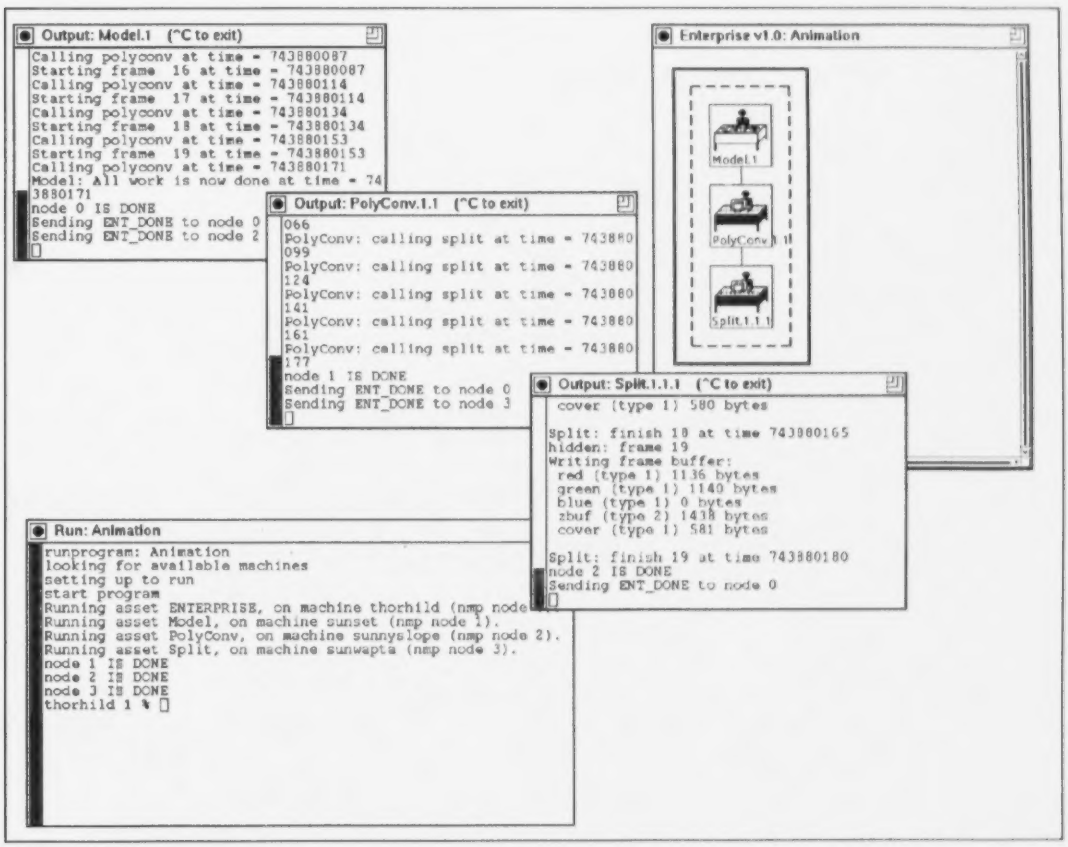

Output: Model.1 (^C to exit)
Calling polyconv at time = 743880087
Starting frame 16 at time = 743880087
Calling polyconv at time = 743880114
Starting frame 17 at time = 743880114
Calling polyconv at time = 743880134
Starting frame 18 at time = 743880134
Calling polyconv at time = 743880153
Starting frame 19 at time = 743880153
Calling polyconv at time = 743880171
Model: All work is now done at time = 74
3880171
node 0 IS DONE
Sending ENT_DONE to node 0
Sending ENT_DONE to node 2
Output: PolyConv.1.1 (^C to exit)
066
PolyConv: calling split at time = 743880
099
PolyConv: calling split at time = 743880
124
PolyConv: calling split at time = 743880
141
PolyConv: calling split at time = 743880
161
PolyConv: calling split at time = 743880
177
node 1 IS DONE
Sending ENT_DONE to node 0
Sending ENT_DONE to node 3
Enterprise v1.0: Animation
Model.1
PolyConv.1.1
Split.1.1.1
Output: Split.1.1.1 (^C to exit)
cover (type 1) 580 bytes
Split: finish 18 at time 743880165
hidden: frame 19
Writing frame buffer:
red (type 1) 1136 bytes
green (type 1) 1140 bytes
blue (type 1) 0 bytes
zbuf (type 2) 1438 bytes
cover (type 1) 581 bytes
Split: finish 19 at time 743880180
node 2 IS DONE
Sending ENT_DONE to node 0
Run: Animation
runprogram: Animation
looking for available machines
setting up to run
start program
Running asset ENTERPRISE, on machine thorhild (nmp node
Running asset Model, on machine sunset (nmp node 1).
Running asset PolyConv, on machine sunnyslope (nmp node 2).
Running asset Split, on machine sunwapta (nmp node 3).
node 1 IS DONE
node 2 IS DONE
node 3 IS DONE
thorhild 1 %

Figure 3.11: Screen display after executing the Animation program

After execution we want to check that the program produced the correct results. The sample program creates files containing animation frames in its current directory. To check the results, we ran a utility program that reads the animation frames and displays them in an X window. This was done from the console window.

3.8. Performance Tuning

When we set the global compile and run options in Section 3.5, we turned on the event logging flag. This caused the program to capture events while it ran and logged them to an event file so the execution can be animated. This allows us to see if there are any performance bottlenecks and to see if the program is making full use of its resources.

Program display is done from the animation view. The view is displayed by selecting Animate from the design view menu. After changing views, the program appears as in Figure 3.12. The view is similar to the design view in Figure 3.6, but now the state of each asset is displayed and there is space to display message queues above the assets and reply queues to their right.

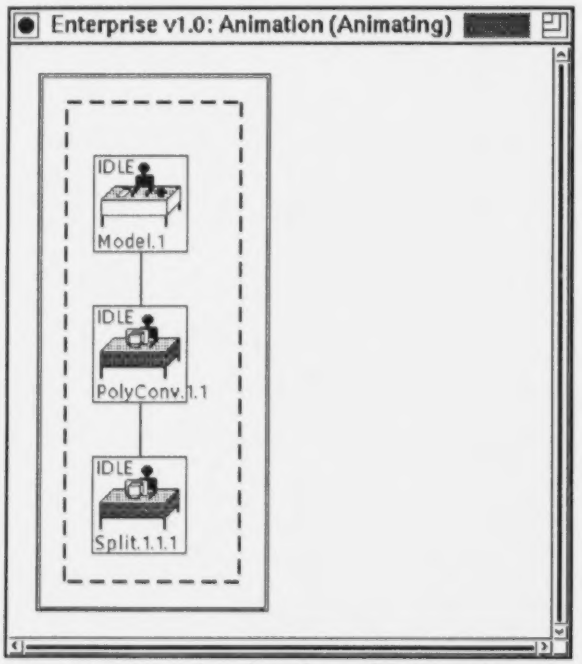


Figure 3.12: The animation view of the sample program

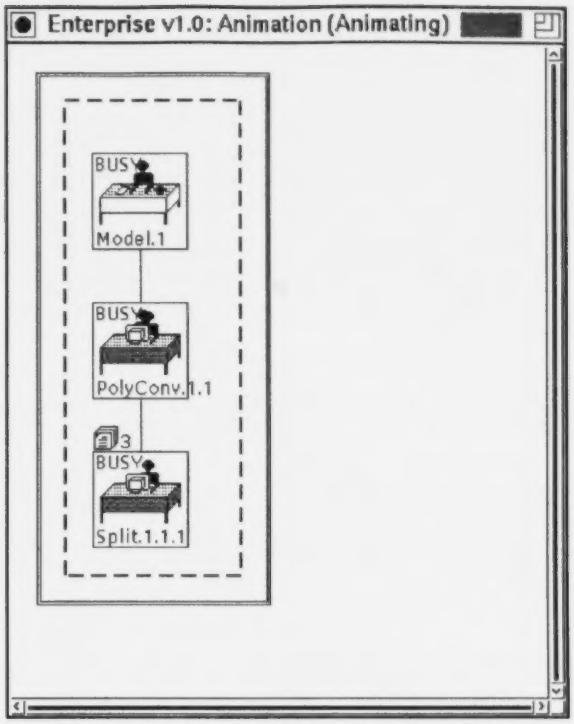


Figure 3.13: A point in the animation

The menus that appear are now different as well. Clicking in the animation view, but outside of the assets brings up the *animation menu*. If we select Start from the menu, the program execution will be displayed, showing messages and replies moving between assets and updating asset states as they change. Eventually, we see that the messages are building up in Split's input queue as shown in Figure 3.13.

Split can't keep up with the amount of work being sent to it by PolyConv. We can try to improve the performance by replicating Split. To do this, we end the animation by selecting Stop from the animation view menu, then switch back to the design view by selecting End. Once we are in the design view, we select Replicate from Split's asset menu, causing the replication dialog box to appear as shown in Figure 3.14.

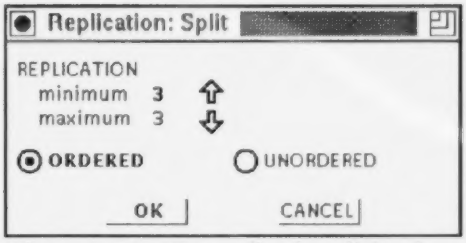


Figure 3.14: The replication dialog box

We replicate Split three times by clicking on the up arrow until the maximum replication factor becomes 3. This gives us three copies of Split when we run the program. Now we can close the box, re-compile, and re-execute the program.

When we switch to the animation view after re-executing the program, we can see the replicas of Split. This time when we display the run, we see that all assets keep busy and no messages build up in Split's input queue as shown in Figure 3.15.

We could try to improve performance further by replicating PolyConv, but when we do this we see that all but one of the replicas are idle most of the time. This indicates that we are wasting resources without improving the performance. We could also try other things like adjusting the replication factors or changing Split into a line to do hidden surface removal and antialiasing in separate steps. None of these changes makes much difference to the performance however.

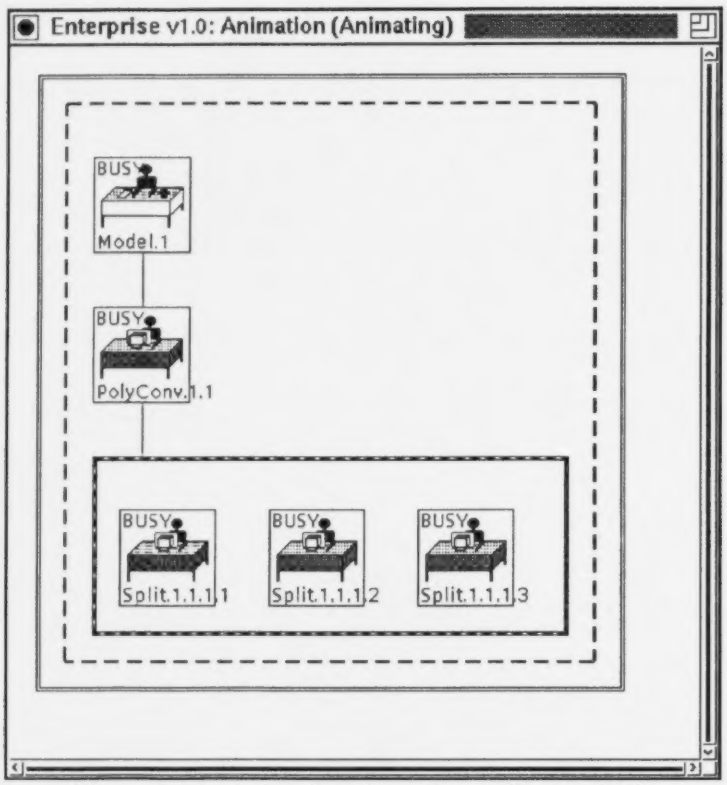


Figure 3.15: Animation view after replicating Split

Chapter 4.

The Enterprise User-Interface

This chapter describes the Enterprise user interface. It consists of a section that describes how to use the interface, followed by a section on how the interface was implemented. The first section defines the system at the point in time that this thesis was written. Time constraints prevented finishing all of the implementation, but the major parts have been implemented and the rest of the features exist in prototype form. Chapter 6 lists the parts of the system described here that have not been completely implemented.

4.1. Using the Interface

4.1.1. Overview

Enterprise is a complete environment for building distributed programs. Enterprise programs are organized into assets as described in Chapter 2, and consist of two main elements, a graphical specification of the parallelism among assets and normal C source code specifying asset behavior. The user-interface provides integrated facilities for editing the parallelism graph, editing the source code, compiling and executing the program, and monitoring the program using animation. Eventually the system will include real-time monitoring and a debugger as well. All functions are invoked from context-sensitive menus linked to graphical objects. The object under the cursor when a menu is invoked determines which menu appears and what operations are available. The operation selected is applied to the object linked to the menu.

The user-interface has been implemented in Smalltalk-80, version 4.0. It may be used to construct programs on any machine supported by Smalltalk-80, including UNIX™ workstations, Macintoshes, and IBM 80x86 or compatible machines. However, since the rest of the Enterprise programming environment is UNIX dependent, some features such as Compile and Execute only work on UNIX workstations. The rest of this chapter discusses the UNIX version of Enterprise, which runs under X windows.

4.1.2. The User Model

As described in Chapter 2, Enterprise represents a distributed program using an analogy with a business organization. Program modules and parallelization techniques are represented by *assets* such as individuals and departments. The assets are displayed as graphical icons in the interface, where the user builds programs by manipulating the assets using menu commands.

If an asset is selected by clicking the middle mouse button, a menu is displayed that contains only those operations that are currently valid for the asset. If no asset is selected when the button is clicked, the *design menu* containing global program operations is displayed. Thus it is impossible for a user to select an invalid operation. This approach simplifies the user's mental model of the programming environment since it reduces the number of operations the user sees [LSW87]. It contrasts with pull-down menus where

---

™ Unix is a trademark of Bell Laboratories

the user is presented with a plethora of choices, some of which have subtle differences and some of which do not even apply to the user-interface component being considered. For example, if the user chooses Compile from an asset's menu, only the code for the asset is compiled. If the user chooses Compile from the design menu, all assets are compiled. Furthermore, the Execute command does not even appear in an asset menu. In a pull down system, Compile Asset, Compile Program, and Execute would all appear in the menus.

The interface uses several windows and dialog boxes. The window title bars identify the kind of window and indicate its status. For example, when there is no program currently being edited the title bar contains 'Enterprise: (unnamed)', but when a program is being edited the name of the program replaces '(unnamed)'. The title bars of dialog boxes indicate the type of dialog box and the name of the asset or program it is linked to.

Hierarchical Assets

Assets can be nested inside each other hierarchically. *Composite assets* such as enterprises, departments, lines, and divisions that contain other *component assets* can be *expanded* to display their components and allow them to be edited, or *collapsed* to hide the components and allow the entire unit to be viewed as a single icon. In either case, the menu of the composite asset is always available. If the asset is collapsed, the menu appears when the icon is selected. If the icon is expanded, the composite asset becomes a rectangle with its components nested inside it and the menu that appears is determined as described below. The border style and the position of the components inside it indicate the type of the asset. Figures 4.1 through 4.4 show the collapsed and expanded composite assets.

When the middle mouse button is clicked, the innermost rectangle containing the cursor determines which menu appears. For example, in Figure 4.3, if the cursor is inside the dashed line rectangle but outside the icons, the menu for the line asset appears. If the cursor is inside the double line rectangle but outside the dashed line rectangle, the menu for the enterprise asset appears. The different borders provide an easy way for the user to determine which asset a border represents when several levels of nesting are visible.

Expanded enterprise assets have a double line as a border. The enterprise asset represents the entire program or enterprise and initially has a single individual as a component. No other components can be added to the enterprise, but the individual can be re-classified as a department, line, or division by making the appropriate selection from its asset menu. Neither the enterprise nor its component can be replicated.

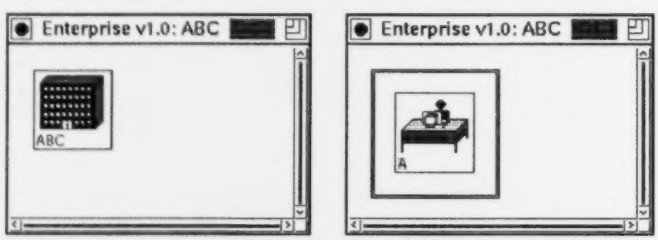


Figure 4.1: Collapsed and expanded enterprise assets

Every composite asset, except the enterprise asset, has a receptionist as its first component. Expanded department assets have a double width solid border and have their

components arranged horizontally below their receptionist. Expanded line assets have a double width dashed border, and their components are arranged vertically below their receptionist. As well as a receptionist, each department and line initially contains one individual as its second component. More components can be added by selecting Add After from the menu for either the receptionist or individual. A new individual will be added after the component whose menu was used. Components can be deleted by selecting Delete from their asset menus. All components except the receptionist can be replicated or re-classified as departments, lines, or divisions by making selections from their asset menus.

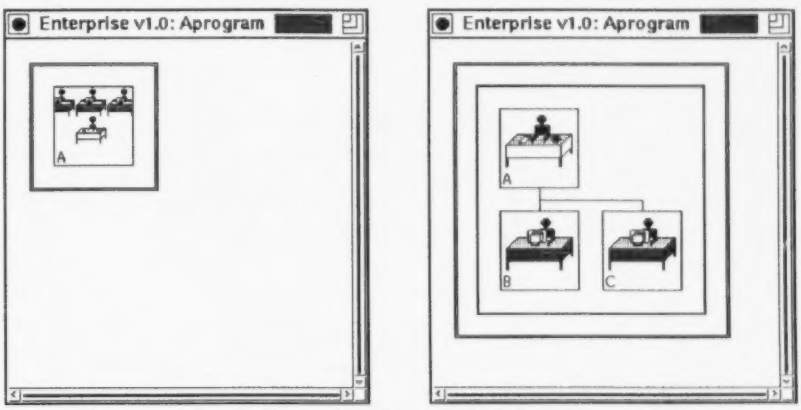


Figure 4.2: Collapsed and expanded department assets

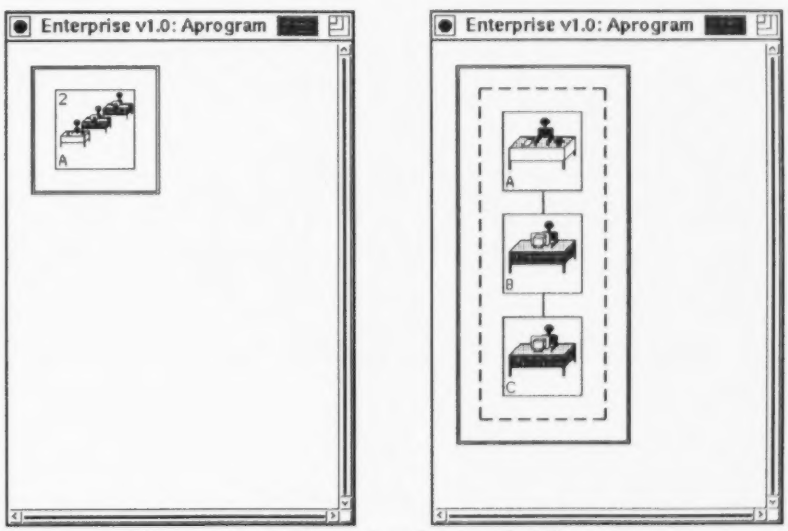


Figure 4.3: Collapsed and expanded line assets

Expanded division assets have a shaded double width border and a single component directly below their receptionist. Initially this component is a representative, which can be replicated or re-classified as a division. As discussed in Chapter 2, divisions are used to represent parallel recursion in Enterprise. The receptionist and the

representative share the same code and have the same name. When the receptionist makes a recursive call, it actually sends a message to the representative. A parallel recursive program is built by constructing a tree of nested divisions with representatives at the leaves.

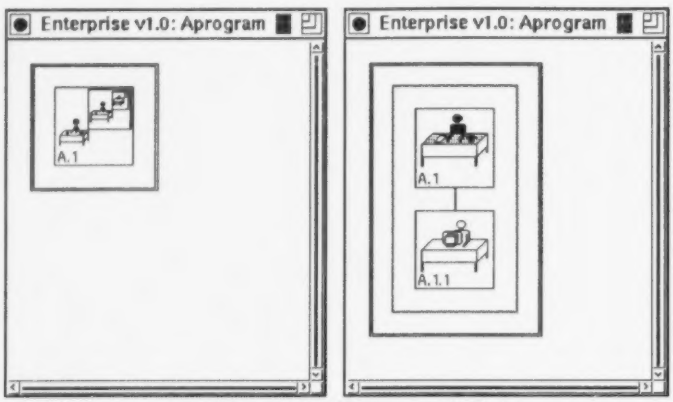


Figure 4.4: Collapsed and expanded division assets

Programming Semantics

The code for each asset must have a function with the name of the asset. The asset is invoked by calling this function. The first asset in the program must take the parameters `argc` and `argv`. It cannot be named `main`.

The asset types impose some restrictions on which asset calls are legal. Each of the composite assets represents a template for a high-level parallelization technique. The compiler restricts the calls that assets can make to ensure that they conform to the template. For example, a line asset represents the pipeline parallelization technique. This means that each asset in the line should perform some work, produce an intermediate result, then pass the result to the next asset in the line. No asset should call any other asset that is not the next one in the line. The compiler enforces this rule.

Any calls that violate the rules are reported at compile time and are treated as fatal compile time errors. This way Enterprise can improve run time performance by setting up the connections between assets before the program first starts.

Departments, lines, and the enterprise asset can never make asset calls because they do not have code associated with them. For other asset types, whether a call is legal depends on the type of the caller's parent. The following restrictions are made on asset calls:

- Services are globally visible and can be called by any other asset, but cannot make any calls themselves.
- Only the receptionist of a composite asset is visible outside its parent. It is the only asset that can be called from outside the composite asset.
- Inside a department, only the receptionist can call the other components. If the department itself is a component of a line, all components of the department,

including the receptionist, can call the next component of the line. Note that if this next component is a composite asset, the call will actually be to its receptionist.

- Inside a line, each component, including the receptionist, can only call the next asset in the line. If the line itself is a component of a second line, the last component can call the next component of the second line.
- Inside a division, both the receptionist and the component division or representative can recursively call the division. Like a department, if the division is a component of a line any of its components can call the next component of the line.

Replication

Most assets can be *replicated*. Replicating an asset creates copies of the asset that execute in parallel, each *replica* using the same source code. The run-time executive keeps track of which replicas are busy and which are idle. When a call is made to the replicated asset, the job will be processed by one of the replicas. If all replicas are busy, the job will wait until a replica becomes free. If enough machines are available, each replica will be placed on a different machine. Otherwise, some may share machines with each other or with other assets.

Replicable assets have minimum and maximum *replication factors* which set a range for the number of replicas of the asset that are created. The minimum replication factor specifies the number of replicas created statically, before the user's program is started. The maximum replication factor specifies the maximum number of replicas that will be created for an asset, including those created dynamically at run-time. When a call is made to an asset at run-time, if all its replicas are busy and it has less total replicas than its maximum replication factor, a new replica of an asset is created dynamically. If the maximum number of replicas are already running, the job waits until one is available. The minimum must be greater than zero, and the maximum must be greater than or equal to the minimum.

Divisions are used to represent parallel recursion. They consist of a tree of nested divisions with representatives at the leaves. The branching factor of the tree is set for a level by replicating the division or representative. When a recursive call is made to a division, it will execute in parallel if a replica is available, or wait until one becomes free.

Receptionists cannot be replicated. If a receptionist was replicated, there would be several entry points to the same composite asset. When a composite asset is replicated all of its components, including the receptionist, are replicated as a unit, giving each composite replica a unique entry point. In both cases the caller would call one of several copies of the receptionist, but in the first case they would all talk to the next component in the same composite asset, whereas in the second case they would each talk to a component in a different composite asset.

Neither the enterprise asset nor its component can be replicated. This is because the current version of Enterprise does not support multiple copies of a program running under one copy of the interface.

There is a subtle difference between the semantics of replicated assets that use static variables and the semantics of a sequential program using the same code. Replicas are not re-created each time they are invoked, but are implemented as a process that starts

up, then waits in a polling loop for incoming messages. When a message is received, the replica processes the job, then goes back into the loop. It does not re-initialize static variables each time. On the other hand, the semantics of sequential C are that all static variables are initialized when the program first starts. Thus, a replica may incorrectly assume that a static variable has been initialized when it actually contains a value from a previous call.

Re-Classifying Assets

The kind of parallelism is determined by the asset kind as described in Chapter 2. An asset's kind can be changed by *re-classifying* it. Initially, a new program consists of an enterprise with a single individual as a component, representing a sequential program. To change the asset kind, the individual must be re-classified as another kind such as a department or line by making selections from the asset menu. For example, to re-classify an individual as a line, the user selects Line from the individual's asset menu. The individual will then be replaced by a line.

Re-classifying an asset causes a new asset of the desired type to be created that replaces the original one in the asset hierarchy. The new asset has all of the original's attributes such as its name and replication factors. New departments and lines are created by re-classifying individuals. They initially have two components: a receptionist which has the same name as the composite asset and an individual which has a default name assigned by the system. Any existing code is transferred to the receptionist. New divisions are created by re-classifying individuals or representatives. Like departments and lines, a division has a receptionist that shares its name, but its second component is a representative rather than an individual. Both of the components of a division have the same name as the parent division.

There are some restrictions on which assets can be re-classified. Receptionists cannot be re-classified because their role is fixed as the entry point of a composite asset. Representatives can only be re-classified as divisions because their role is fixed as the leaves of a division tree. Services cannot be re-classified as any other asset kind, and no other assets can be re-classified as services. Individuals can be re-classified as departments, lines, or divisions. Divisions that have another division as a parent can only be re-classified as representatives. Other composite assets, including divisions whose parent is not a division, can only be re-classified as individuals.

When a composite asset is re-classified as an individual, all of its components are deleted and a new individual with the same attributes is created to replace it. The code from all of its components is concatenated into one file for the new individual and the files for the components are deleted. The user is warned before these actions are taken.

4.1.3. Starting Up

Running the Interface

Assuming that the interface has been installed as described in Appendix C and the Smalltalk image has been saved with the default name of `st80`, start Smalltalk by changing to the Enterprise directory and executing the UNIX command `'st80'`. Once Smalltalk has started, evaluate the expression `'Enterprise open'` to start the interface. This can be done by typing it into a workspace, highlighting it, and then selecting doit from the middle mouse button menu. If there is no workspace, open one by selecting workspace from the

Launcher Utilities menu. A window should then open that displays an enterprise asset as shown in Figure 4.1.

Initial Menus

To begin building a program, you must first either select an existing program to edit or create a new one. This can be done by using the design menu or the asset menu for the enterprise asset.

The design menu contains the choices Edit Program and New Program. Selecting Edit Program will display a list of existing programs to choose from. Clicking on the desired program will load it for editing. Clicking outside of the list will cancel the operation. Selecting New Program will cause a dialog box to appear asking for a name. Type a name in the box and press ENTER. The interface checks that the name is unique, then creates the required directories and files.

The asset menu contains the single choice Name. Selecting it will display a dialog box asking for the name of a program to edit. Typing the name of an existing program will load it just as though the program had been selected from the list displayed by Edit Program in the design menu. Typing a name that does not already exist causes a new program to be created with that name just as if New Program had been selected from the design menu. To avoid creating a new program by mistake, it is safest to load an existing program from the design menu.

Once a program has been created or loaded, the menus change to include more choices and the title bar of the window changes to include the name of the program. The enterprise asset menu will still contain the choice Name, but selecting it will now rename the program instead of creating or loading one.

4.1.4. The Design Menu

The design menu contains the following choices:

New Program	Create and edit a new program.
Edit Program	Load and edit an existing program.
Save Program	Save the current program.
Delete Program	Delete the current program.
Edit User File	Edit non-asset C source, header, or data files.
Compile All	Re-compile all assets and source files.
Make	Re-compile changed assets and source files.
Execute	Execute the program.
Animate	Switch to the animation view.
Set Options	Set global compile and run options.
Edit Machine File	Edit the list of available machines.
Show Services	Display the service canvas.

New Program

This command is used to create a new program. Selecting this command causes a dialog box to appear asking for a program name. Pressing ENTER with no name in the box cancels the operation. Typing a name in the box and pressing ENTER creates a new program and readies it for editing. If the name is in use by a program, an error message is displayed and the operation is aborted.

After the name has been entered, Enterprise creates sub-directories for the program as described in Section 4.2. It then saves the current program, creates a new enterprise asset, and replaces the current enterprise asset with the new one. The name of the new program appears in the window title bar. Because of the implicit save, the user is not warned that the current program will be replaced. It can be re-loaded simply by selecting Edit Program.

Edit Program

This command is used to load an existing program for editing. Selecting this command causes a list of existing programs to appear. Selecting one of the programs by clicking on it causes the current program to be saved and the new program to be loaded. Clicking outside of the list will cancel the operation. The new program replaces the current program in the window and its name is displayed in the title bar.

Save Program

This command is used to explicitly save the current program. A representation of the asset hierarchy is written to a *graph file*. The graph file is used to save and re-load programs, as well as to communicate between the interface, pre-compiler, and executive. Its format is described in Appendix A.

The file is written in the Graph sub-directory of the program's directory. Its name is the name of the program with the extension .graph appended to it.

Currently there is no Save As facility for saving the program with a different name. Instead, the user must first rename the program by selecting Name from the enterprise asset's menu, then select Save from the design menu.

Delete Program

This command is used to delete a program. Selecting this command causes a dialog box to appear asking the user to confirm the operation. Selecting NO aborts the operation. Selecting YES causes the current program to be deleted. All of its files are deleted including its asset code and graph file. The program cannot be recovered afterward.

The interface will return to the state where no program is being edited, as was described above.

Edit User File

This command is used to edit files containing anything other than asset source code. Asset source code is edited by selecting Code from the asset's menu. Selecting Edit User File causes a sub-menu to appear with the choices, Edit Source File, Edit Header File, and Edit Data File.

Edit Source File edits code that is shared by assets. Because all assets are linked together into one executable program, common code must not appear in more than one asset. It is sometimes easier to put common code into a separate file that is then linked with the assets. For example, consider an individual asset containing functions `Read()`, `Sort()`, and `Write()` that all call another function named `DisplayTime()`. If the individual was re-classified as a line with component assets `Read`, `Sort`, and `Write`, the code for `Read()`, `Sort()`, and `Write()` would all be put into files for their associated assets. `DisplayTime()` could be put into a separate file. The files are placed in the User

sub-directory of the program directory and are automatically compiled and linked with the program. They are not pre-compiled, so they cannot contain asset calls.

Edit Header File allows header files to be edited. They are placed in the Include sub-directory of the program directory. The compiler searches this directory when processing `#include` directives.

Edit Data File edits input data files used by the program at run time. The file name can be specified as an input file in the run-time options section of the Set Options dialog box described below. Data files are placed in the Data sub-directory.

All three selections display a dialog box containing a list of existing files of the appropriate type as shown in Figure 4.5.

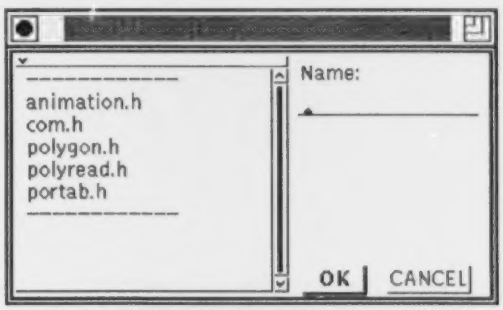


Figure 4.5: Dialog box for selecting a file to edit

Selecting the CANCEL button aborts the operation. Selecting a file from the list, then selecting OK will start an editor containing the selected file. Typing a name in the Name box, then selecting OK will create a new file and start an editor for it. If an editor was specified in the .ENTrc file, it is used. Otherwise a standard Smalltalk editor is used.

Compile All

This command is used to compile and link the entire program. The source code for all assets is pre-compiled, the resulting files are compiled, and the files in the User directory are compiled. Then the resulting object modules are linked with Enterprise system modules, Enterprise system libraries, and user specified libraries to produce an executable program. The user specifies libraries in the Set Options dialog box described below. The current program is saved before compiling to ensure that the graph file is up to date for the pre-compiler.

When the compiler is running, the cursor changes to indicate that Enterprise is busy. The interface will be inactive until the compilation is finished. Pressing CTRL-C will abort the compile.

A window is opened that shows the status of the compilation. All compiler and linker messages, including error messages, are displayed in the window. When the compilation is done, the window is left open with a UNIX shell running in it. The user can enter commands or scroll the window back to view error messages. Pressing CTRL-D exits the shell and closes the window.

Any part of the asset hierarchy can be compiled without linking by selecting Compile from the asset menus. The asset and all of its components will be compiled as described below.

Make

This is similar to Compile All, but only the code that has changed since the last compile is re-compiled. It has a functionality similar to UNIX `make`, except it does not consider header files and the user has no control over dependencies.

Run

Once a program has been successfully compiled and linked, Run can be used to execute it. The current program is first saved to its graph file, then the program is started in its directory. While the program is running, the cursor changes to indicate that the interface cannot be used until execution is finished. Pressing CTRL-C aborts the run.

If the program has not been compiled or if the executable is older than the source code for an asset, a dialog box appears. The box contains the choices None, Old, and Doit. Selecting None will abort the run. Selecting Old will execute the out-of-date executable. Selecting Doit will re-compile the program, then run it.

A console window is opened that contains messages from the Enterprise run-time executive. Like the compile window, it is left open so the user can review system messages or enter UNIX commands to test the result of the run. If output or error window options were enabled for any assets, windows containing the standard output and standard error files for the assets would also appear. These types of asset specific options are set using the dialog box invoked by selecting Set Options from the asset menus. Global program options such as command line arguments or input data file names are set using the dialog box invoked from the design menu. The boxes are described below.

Animate

This command is used to display the program's last execution. Selecting Animate causes the design view to be replaced by an *animation view* where replicated assets are expanded to show all their replicas, asset states are displayed, and message queues are visible. If the design view is thought of as a view from in front of the asset graph with all but the first replica hidden, the animation view can be thought of as a view from the side of the graph with all replicas visible. When the animation view is visible, the choices available in the menus change to include animation commands.

The execution is displayed using an event file produced during program execution. The event logging flag must be turned on in the program's Set Options dialog box before events are captured.

Chapter 5 describes the animation system, including the animation view, in more detail.

Set Options

This command is used to set compile and run options that affect the program as a whole. Options that affect single assets or a part of the asset hierarchy are set from the asset menus as described below. The options are not saved between sessions, but are reset to their default values every time the interface is started or the current program changes.

Selecting Set Options causes the dialog box shown in Figure 4.6 to appear. Compile options appear in the top half of the box, and run options are in the bottom half.

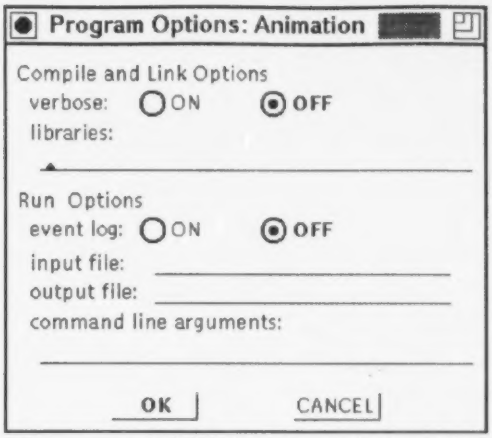


Figure 4.6: Dialog box for global options

The available compile options are the verbose flag and a specification of external link libraries. The verbose flag determines how much information the pre-compiler displays in the compiler window as it works. Turning the flag on causes more output to appear. Most of the extra output is only useful for debugging the pre-compiler itself. The libraries box specifies options for the linker. Whatever is typed into the box is appended to the link command by the compiler. For example, to have the program linked with the math library and a graphics library named `libUTILITY.a`, `'-lm -lUTILITY'` should be typed in the box. Enterprise adds the User directory to the compiler's library search path.

The run options are the event logging flag, input and output file names, and command line arguments. Turning on the event logging flag will cause events to be captured and saved to an *event file* as the program runs. The file will be used to display the execution using Enterprise's animation component as described in Chapter 5. The format of the file is described in Appendix B. Entering the name of an input file or output file will cause the first asset in the program to read its standard input or standard output from the file. The contents of the command line box will be passed to the first asset in the program when it starts.

Once all the fields in the box have been set as desired, pressing the OK button or pressing ENTER will set the options. Pressing the CANCEL button will discard the contents of the box and leave the options as they were set before it was opened.

Edit Machine File

This command is used to maintain the list of machines that are available to run an Enterprise program. Enterprise maintains a *machine file* for each program that determines the machines that are available. When Edit Machine File is selected, the machine file editor window shown in Figure 4.7 appears.

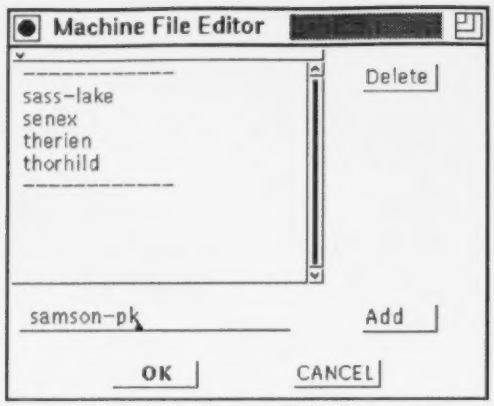


Figure 4.7: The machine file editor

The list contains the current set of machines. A new machine can be added to the list by typing its name into the box below the list and pressing the Add button. The machine name will be added to the list and disappear from the Add box. The name should be one that can be used in a UNIX `rsh` command. The interface does not check that the name is valid. A machine can be deleted from the list by selecting it in the list, then pressing the Delete button.

Pressing the OK button will update the machine file and close the box. Pressing CANCEL will close the box but ignore its contents, leaving the machine file unchanged.

The set of machines used to run a given asset can be overridden using the EXCLUDE and INCLUDE options in its options dialog box described below.

Show Services

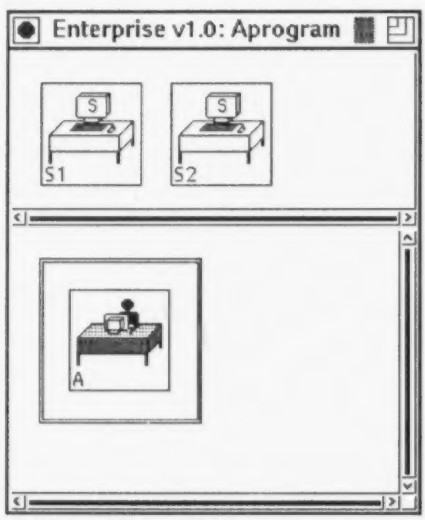


Figure 4.8: The service canvas

This command is used to display the service canvas. When the interface first starts, the program canvas fills the entire window. The service canvas exists, but it is

hidden. Selecting Show Services causes it to appear in the top half of the window as shown in Figure 4.8.

If the program currently contains services, they will appear in the service canvas and their menus can be selected like any other asset. Both canvasses can be scrolled independently.

The Show Services menu choice will disappear and be replaced with the choices Add Service and Hide Services. Add Service will add a new service to the program, causing a new service icon to appear in the service canvas. Hide Services will cause the service canvas to be hidden once again and the menu will change back to its original contents.

4.1.5. Common Operations on Assets

Several operations that are common to all assets are discussed here. Not all of them appear in the menu for every asset; only those that are legal for the asset at its place in the graph appear.

Naming

When an asset is created, Enterprise gives it a default name. This name can be changed by selecting Name from the asset's menu. A dialog box will appear containing the current name. Typing a new name and pressing ENTER renames the asset. Deleting the name and pressing ENTER will cancel the operation. It is impossible to have an asset without a name.

The name of the asset must be unique within the program or an error message will appear and the operation will be aborted. No asset can be named `main`. There is no limit on the length of names, but shorter names fit inside the asset icons better and look better on the display.

A department or line shares its name with its receptionist. Naming a receptionist also names the parent, and naming the parent also names the receptionist. A division shares its name with both its receptionist and its component representative or division. This means that all elements of a division tree share the same name. Naming any receptionist, representative, or division in the tree names them all. The full name of an asset in a division consists of this *base name* and a numerical suffix appended by the system, giving each element of the tree a unique name. The algorithm used to generate the suffix is described in Chapter 5.

Expanding and Collapsing

As discussed previously, a composite asset can be expanded to reveal its components to make them accessible for editing, or collapsed to view it as a single unit. An expanded asset displays itself as a rectangle containing its components. The border style of the rectangle represents the type of the composite asset. The menu for the composite asset can still be accessed by clicking inside the rectangle but outside of any component.

The menu of a collapsed composite asset contains the Expand command. Selecting the Expand command expands the asset and changes the menu to contain the Collapse command instead of Expand. Selecting Collapse will collapse the asset and replace its menu entry by Expand.

Collapsing assets can be used as a form of clustering [Tay92]. Assets can be collapsed and expanded to control the amount of detail displayed at any place in the asset

graph. This is very useful in the animation view where all replicas are visible and the user may only be interested in a small part of the graph.

Adding and Deleting

Departments and lines are created with a receptionist and one individual as components. More components can be added by selecting Add After from the asset menu for one of the components. A new individual is added to the department or line after the asset whose menu was used. Services also have Add After in their menus. A new service can be added by selecting Add After from the asset menu of an existing service or by selecting Add Service from the design menu when the service canvas is visible.

If an asset can be deleted, the choice Delete will appear in its menu. Selecting it will cause a dialog box to appear asking for confirmation. Clicking on the YES button will delete the asset and all of its associated files. Clicking on the NO button will abort the operation.

Components cannot be added to or deleted from divisions.

Replicating

If an asset can be replicated, the choice Replicate will appear in its menu. Selecting Replicate will cause the replication dialog box shown in Figure 4.9 to appear.

To set the replication factors, click on min or max, then click on the arrows to increment or decrement the selected value. If min is equal to max, incrementing min will also increment max. If max is equal to min, decrementing max will also decrement min. Neither can be decremented to less than 1. To set the ordering option, click the ORDERED or UNORDERED radio buttons.

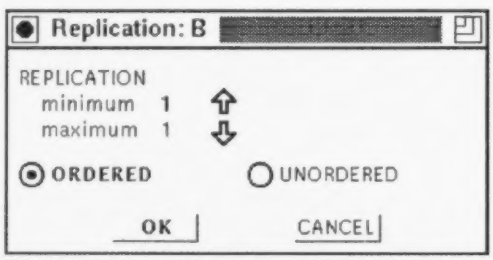


Figure 4.9: The replication dialog box

Clicking on the OK button or pressing ENTER will update the asset's attributes using the values set in the box. Clicking CANCEL will abort the operation and leave the asset unchanged.

Re-Classifying

Assets that can be re-classified have the names of other asset kinds in their menus. The names that appear depend on the asset that owns the menu and its location in the graph. For example, divisions can be re-classified as individuals if they are the root of a division tree, but must be re-classified as a representative if they are an internal node. The choice Individual appears in the menu of a root division. The choice Representative appears in the menu of an internal division.

As discussed previously, re-classifying an asset by selecting the name of another asset kind from its menu causes a new asset to be created that replaces the original one. If

a composite asset is being re-classified as an individual, a dialog box will appear warning that information is being re-organized and asking for confirmation.

Editing Code

Assets like individuals and services that have code associated with them have the choice Code in their menus. Selecting this command causes an editor to open on the code for the asset.

If no code currently exists, a new file is created. The new file will have the same name as the asset, with the extension '.e'. It will be placed in the Assets sub-directory of the program's directory. If code does exist, it will be loaded into the editor. Editors can be open on many assets at the same time. If the Smalltalk editor is used, code can be copied between editors.

The editor that is used is specified in the .ENTrc file as explained in Appendix C. If no editor is specified, a standard Smalltalk editor is used. Details on using it can be found in the Smalltalk documentation [PP90].

Setting Compile and Run Options

Each asset has the choice Set Options in its menu. This command allows compile and run options to be set for the asset. When Set Options is selected, the dialog box shown in Figure 4.10 appears.

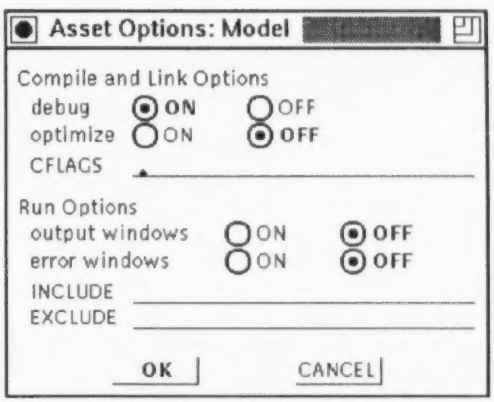


Figure 4.10: Asset Options Dialog Box

If the asset has components, the options are set for all its components as well, overwriting any options previously set. Alternate options for the components can then be set individually from their own menus.

When a program is compiled, Enterprise first inserts code for communication and synchronization by precompiling the assets. It then invokes a standard C compiler on the result. The compile options are for the standard C compiler and consist of a debug flag, an optimize flag, and a string of compiler options. The two flags are compiler independent. If the debug flag is ON, the asset will be compiled using the compiler's debug option so that a standard debugger can be used. If the optimize flag is ON, the asset will be compiled using the compiler's optimization. The CFLAGS box may be used to enter any other compiler options that the user desires. They are not examined in any way by the interface, but are passed directly to the compiler. It is the user's responsibility to ensure

that they are valid options for the compiler used. The flags are most useful for entering such things as macro definitions. For example, an asset could have some code that is included only if the symbol TEST is defined. The string '-DTEST' can be entered in the CFLAGS box to have the compiler include the code.

The run time options specify whether output windows appear, as well as lists of machines to use or avoid when running the asset. If the output windows option is ON, a window is created for the asset when it runs. The window will contain everything the asset writes to its standard output file during the run. Similarly, if the error windows option is on, a window for the asset's standard error output is created. The INCLUDE and EXCLUDE boxes are used to specify machines. They override the list in the machine file described previously. Any machines listed in the INCLUDE box will be used to run the asset. If they are not available the asset will not be started; no other machines will be used. Any machines listed in the EXCLUDE box will not be used to run the asset. These options can be used for such things as forcing an asset to run on a specific graphics workstation or a workstation with a large memory capacity.

Like the other dialog boxes in the system, clicking on the OK button or pressing ENTER will use the settings in the box to update the assets. Clicking on CANCEL will abort the operation and leave the asset unchanged.

Compiling

This command is used to compile an asset and all its components without invoking the linker. The asset will be pre-compiled and compiled. No code in the User directory is compiled and no linking is done. If the asset has components, all the asset's components will be compiled.

4.1.6. Asset menus

Command	Individual	Receptionist	Representative	Department	Line	Division	Service	Enterprise
Name	√	√	√	√	√	√	√	√
Expand/Collapse				√	√	√		√[4]
Add After	√[1]	√		√[1]	√[1]	√[1,2]	√	
Delete	√[1]			√[1]	√[1]	√[1,2]	√	
Individual				√	√	√[2]		
Representative						√[3]		
Department	√							
Line	√							
Division	√		√					
Code	√	√	√			√	√	
Compile	√	√	√	√	√	√	√	
Replicate	√[1]		√	√[1]	√[1]	√[1]		
Options	√	√	√	√	√	√	√	

1- only if the parent of the asset is not an enterprise

2- only if the parent of the asset is not a division

3- only if the parent of the asset is a division

4- only if a program is currently being edited

Table 4.1: Menu choices for assets

Table 4.1 shows which menu choices appear in the menus of each asset kind. Note that when no program is currently being edited, the enterprise asset menu contains the single choice Name. Also, neither the enterprise asset nor its component can be deleted or replicated, and no more components can be added to the enterprise asset. Thus Add After, Delete, and Replicate will not appear in an asset's menu if it is the single component of the enterprise asset.

4.2. Implementation

4.2.1. The Control Model

Under X Windows, Smalltalk runs as a single process. Since a program may display many Smalltalk windows, the Smalltalk interpreter polls the windows, asking each in turn if it wants control. The default behavior is that a window takes control whenever the cursor is inside it.

The Model View Controller (MVC) paradigm [LP91] is used where the model is an instance of class Enterprise, the view is an EnterpriseWindow, and the controller is an EnterpriseController. The EnterpriseController behaves exactly the same as a default Controller except when the program is animated, which will be described in Chapter 5.

The model is responsible for knowing its enterprise (program). The window is responsible for displaying the enterprise using the values stored by the model. Views are composite objects that can contain sub-views, but the location and size of a sub-view within its parent view is maintained by a wrapper object. That is, sub-views are contained in wrappers, which are themselves contained in a parent view. An instance of EnterpriseWindow contains two wrapped sub-views: an Enterprise view and a Service view. The Enterprise view displays the enterprise and the Service view displays the service assets used by the enterprise. As described in Section 4.1, the Service view can be hidden when it is not used.

When a mouse button is pressed, the window passes control to the view that contains the cursor. The view then determines which asset, if any, was selected. The selected asset is one whose rectangular bounds contain the cursor point. However, since assets may be nested in a hierarchical structure, many assets may contain the point. The selected asset is defined as the innermost one that contains it. Once the selected asset has been determined, it takes control, displays its menu, and takes the appropriate action.

Assets are responsible for knowing their locations, their parent asset, and the other assets they contain. As assets are expanded and collapsed, their locations change and must be updated. Determining the selected asset must take this into account.

The following naive approach for determining the selected asset was tried first. The Smalltalk-80 implementation of MVC provides a default behavior that passes control to the innermost view that contains the cursor. This is implemented by maintaining a list of controllers of currently active windows. Each of the controllers in the list is sent a message in a polling loop. If a controller's view has the cursor, it takes control and asks its view if one of its sub-views wants control. If one does, the controller gives control to the sub-view's controller, otherwise it keeps control itself. The sub-view's controller behaves the same way. Thus, the controller for the innermost sub-view that contains the cursor gets control. Since assets are views, the method that determines if a sub-view

wants control was re-implemented. This was necessary since Smalltalk assumes that sub-views are always displayed. If an asset is collapsed or has no components, the method returns the asset itself. Otherwise the method invokes the original method that recursively finds a component of the asset that wants control.

Unfortunately, this approach failed. There were times when the wrong menus would be displayed. Clicking on an asset would bring up the menu for one of its components, its parent, or even one of its parent's parents. Clicking at the same location again would sometimes display the same menu, but would sometimes display the right one or a completely different one. It seemed like the wrong controller was taking control. This behavior was caused by two different phenomena. First, each asset asked the cursor for its location. When the cursor was moved between the times that two assets queried it, each would receive a different point. Second, the control method was not actually as simple as described previously. The method in the controller that asks sub-controllers if they want control is sent from a loop. The loop iterates until the asset no longer has the cursor. When a menu was displayed, the active control loop was initiated from a controller in a loop that was initiated from a controller in a loop, etc. When the active controller finished processing the user's choice, that loop would not end. The next time the user clicked the mouse the controller would assume that it had the cursor (because it was active), would find that no sub-view wanted control, and would thus take control and display its own menu.

Our second approach alleviated this problem. The Enterprise view determines the coordinates of the cursor and asks the enterprise which asset should be selected. The selected asset is then told to display its menu and perform the appropriate action. The enterprise or any other asset uses the cursor point to determine the selected asset as follows: If the point is outside its bounds it answers nil. If the point is inside its bounds and it does not contain any component assets or it contains component assets but they are not currently displayed, then it returns itself. Otherwise, the asset asks each of its component assets in turn to identify the selected asset until one answers an asset or all respond with nil. The asset then returns this result. Before asking each component asset, the asset asks the wrapper of the component to change the coordinates of the cursor point to the local coordinates of the component. This way the cursor coordinates are only determined once. Also, the menu for an asset is invoked by the Enterprise view directly, not from inside of nested control loops.

4.2.2. Drawing Assets

When an asset receives a display message, it draws itself. Any asset that contains component assets can be either collapsed or expanded. Assets that are collapsed or do not have components are displayed in the same way. First the asset draws its icon. Then it displays its name in the lower left corner of the icon. If the asset is replicated, the replication is indicated by drawing lines above and to the right of the icon to simulate a stack of icons, and by displaying the number of replications outside of the top right corner of the replication lines.

An expanded asset first draws a rectangular border. The size of the rectangle is computed by asking each component for its size and adding room for space between the components. Next, a display message is sent to each component so that it draws itself.

The parent asset then draws the connections between the components. Finally the replication is indicated in the same way as it is for collapsed assets.

The basic drawing behavior is implemented in the Asset class and each Asset subclass provides a method for drawing its own icon. In addition, different line styles are used for the borders of expanded assets as discussed in Section 4.1. The method that draws the border is overridden in these assets to use the appropriate behavior. Similarly, the method that draws connections is overridden to draw the appropriate connections for the various Asset sub-classes.

4.2.3. Communicating with Other Components

Although the user-interface is implemented in Smalltalk, the other two Enterprise system components are implemented in C. The user-interface communicates with the pre-compiler and the executive through UNIX pipes and text files. This section describes the technique for connecting to the external UNIX processes, the organization of the directories containing C source and object code files for a program, and several kinds of text files that are used to communicate with the other Enterprise components.

Graph, Event, Preference, and Machine Files

A *graph file* describes a single Enterprise program. It specifies the hierarchical structure of the assets, replication factors, compile and link options, and any user machine preferences. The assets are listed in a depth-first order. This makes it simple to write the file by traversing the asset hierarchy, and allows the hierarchy to be rebuilt by reading the file sequentially. For each asset there is a line with its name, type, replication factor and options for ordering, debugging and optimization. If the asset has internal components there is also a count of components. Following this are four lines that specify the compile, link and run options. If the asset has components, these lines are followed by their descriptions in the same format. Appendix A contains a description of the Enterprise graph file format.

Graph files are created and edited by the user-interface. When the user selects the Save, Compile, or Run commands from the design menu, the enterprise is asked to store a representation of itself in a graph file whose name is the enterprise name with a .graph appended. Each asset type knows how to write a description of itself and if it has components, it asks its components to write themselves as well. Alternately, when the user wants to load a previously saved program, the graph file is read and as it is parsed, assets are created and displayed to represent the saved program.

The pre-compiler uses the information contained in a program's graph file to identify procedure/function calls to assets and replaces them with message sends and receives. The runtime executive uses the graph file to determine how many processes to launch, the execution role of each process and the appropriate communication links between these processes.

Event files are created by the run-time executive's monitor process while a program is running and are used later to animate the program. The event file name is the name of the enterprise with a .ev appended to it. The events contained in the file are described in more detail in Chapter 5. Appendix B contains a description of the Enterprise event file format.

Enterprise maintains a preferences file. When the user-interface first starts, it looks in the current directory for a file named .ENTrc. The file is read and global preferences are set from its contents. Appendix C includes a list of the preferences currently supported.

Each program also has a *machine file* located in its Graph sub-directory. The file contains a list of machines that the run-time executive can use. Each line of the file contains the name of one machine. The file can be modified by the user from the design menu by selecting Edit Machine File.

External Processes

The user-interface launches external processes for compiling code, running a program, and possibly for editing code. The user may use a standard Smalltalk editor or a non-Smalltalk editor may be selected. Several editors can be active at the same time, one for each asset. If the Smalltalk editor is used, no new process is launched. Instead, a new Smalltalk window is created and the window is added to the list of active windows. It is given control by the Smalltalk interpreter whenever its window has the cursor. If an external editor is used, an X window is created. The editor becomes an X windows task that executes concurrently with the Smalltalk interpreter.

The Compile and Run commands are only usable with the UNIX version of the user-interface since the pre-compiler and executive currently require UNIX. Both commands launch an external process and establish communications with it. Smalltalk simplifies this task by providing a UnixProcess class. A message is sent to this class specifying the name of a UNIX program, an array of arguments for the command, and a block. The block is evaluated with the external process as an argument. This provides a mechanism for referencing the process from Smalltalk after it has been created. When the message is sent, the process is created and two pipes are established, one connected to the process' standard input and the other connected to both its standard output and standard error. These pipes are represented as Smalltalk streams that are contained in the instance of ExternalConnection that is returned by the message.

The user can elect to compile and link the entire program or to compile part of the asset hierarchy. In either case, if the program has been changed, the user-interface first writes out the graph file. The Enterprise pre-compiler process is then started and a window is created to display all text that is sent to the ExternalConnection's output stream. The event polling loop in the controller for the Enterprise view monitors the stream. Whenever new text is available, it is displayed in this window. If there is no new text, the polling loop just continues normally. The user can interact with the system normally and may even cancel the compile. When the compile is finished, the window is left open so that the user can review the compiler messages. Programs are run in a similar manner except output is displayed in another window.

4.2.4. Sub-Directories

The Enterprise interface runs in the directory created during installation. Two sub-directories are also created:

Bin This directory holds UNIX scripts and executable programs used by the pre-compiler and run-time executive. Some of the scripts are used by the interface

as well. For example, when Run is selected from the design menu, the script Bin/runprogram is called to execute the program.

Lib This directory holds Enterprise system libraries and object modules that are linked with the compiled asset code in Obj and the compiled modules in User.

Whenever a new program is created, Enterprise creates a new sub-directory for the program. The directory has the name of the program with '.A' appended to it, and contains the following directories inside it:

Assets This directory holds the C source code files for all of the assets. Each asset's code is stored in a file ending with .e. The pre-compiler parses these files and produces corresponding files ending with .c. These are written to the Src directory. For example, the user's code for an asset named Model will be stored in Assets/Model.e. The pre-compiler produces the corresponding file Src/Model.c.

Data This directory holds input data files for the program. When Edit Data File is selected from the design menu, the file is written here. The file can be specified as an input file in the options box invoked from the design menu.

Err This directory holds the assets' standard error output. Each asset is assigned a unique file name in the directory. When it runs, anything an asset writes to its C standard error file goes into the file for the asset. The file is rewritten each run. If an error window has been enabled for an asset, the file is copied to the window after the run.

Graph This directory holds system generated files like the graph file, the event log, and the machine file for a program.

Include This directory holds C header files. When the compiler processes #include directives, it searches this directory. When Edit Header File is selected from the design menu, the files are written here.

Out This directory holds the assets' standard output. Each asset is assigned a unique file name in the directory. When it runs, anything an asset writes to its C standard output file goes into the file for the asset. The file is rewritten each run. If an output window has been enabled for an asset, the file is copied to the window after the run.

Obj This directory holds object modules containing the compiled asset code. These are .o files that are linked together with libraries to form the final program.

Src This directory holds pre-compiled C source code for the assets. These are the user's code with the calls to the Enterprise system libraries inserted into them. The files here are compiled using a standard C compiler, producing object modules that are stored in the Obj directory.

Tmp This directory holds temporary files used by the pre-compiler.

User This directory holds C source code for user modules that are to be linked to the asset code. Common routines called by all assets can be placed into files

here and they will be compiled into object modules and linked with the assets. When Edit Source File is selected from the design menu, the file is written here. This directory is also added to the linker's library search path.

As well as these directories, the program's directory holds the executable program itself.

4.2.5. The Asset Inheritance Hierarchy

The section above described the way that assets are drawn and the approach relied heavily on inheritance. In fact, inheritance is used extensively throughout the user-interface, but the asset hierarchy can be used to illustrate its importance. The asset kinds form a natural inheritance graph as shown in Figure 4.11. A solid triangle in the upper left corner of a class denotes an abstract superclass as described in [WWW90]. The abstract class, Asset, is the root of the inheritance tree. Universal responsibilities like naming are defined and implemented in this class. Below the Asset class is a level of abstract superclasses that define several responsibilities that are shared by several of the leaf asset classes. A CodableAsset has an external file of C source code associated with it which can be edited and compiled. A ReplicableAsset can be replicated and transformed to an asset of a different type. A DeletableAsset can be deleted from its parent asset. An ExpandableAsset has component assets so it can be expanded or collapsed. An AddableAsset can have components added to it after it has been created.

The rest of the asset classes are concrete subclasses. A ReceptionistAsset has code, but can't be replicated, deleted, or expanded. A RepresentativeAsset has code and can be replicated but can't be deleted or expanded. An IndividualAsset is like a RepresentativeAsset, except that it can be deleted. A DivisionAsset is like an IndividualAsset, except that it can be expanded. A ServiceAsset has code and can be deleted, but it can't be replicated or expanded. A LineAsset or DepartmentAsset can be replicated, deleted, or expanded, but has no code. An EnterpriseAsset is expandable, has no code, can't be replicated and can't be deleted.

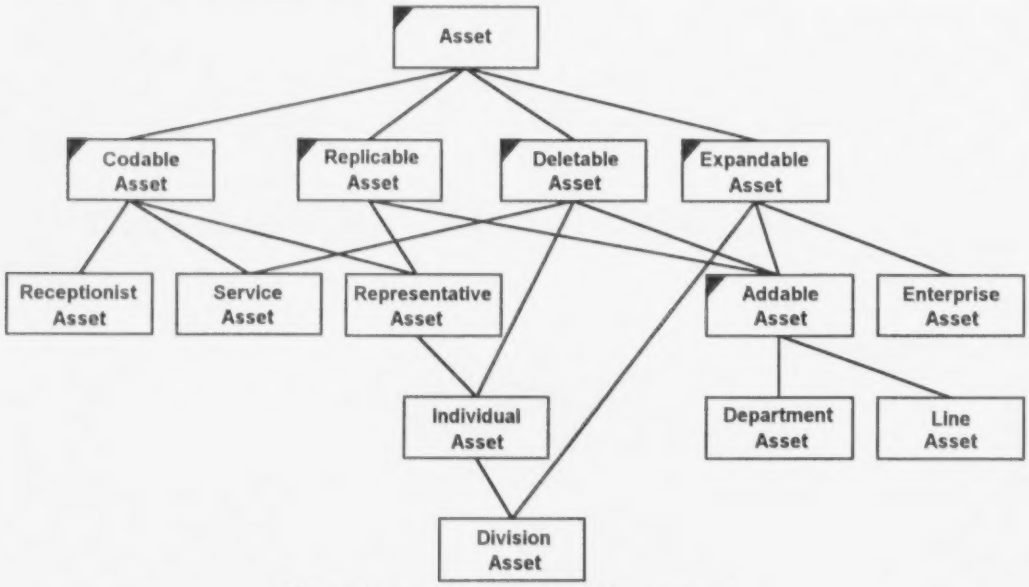


Figure 4.11: The asset inheritance graph

Unfortunately, Smalltalk is restricted to tree inheritance, which required several compromises to be made in transforming this inheritance structure to a tree. The result is shown in Figure 4.12. A comparison of Figures 4.11 and 4.12 illustrates clearly that support for multiple inheritance is essential for applications with real-world models. The lack of multiple inheritance was the most difficult obstacle that needed to be overcome in using Smalltalk for the Enterprise project.

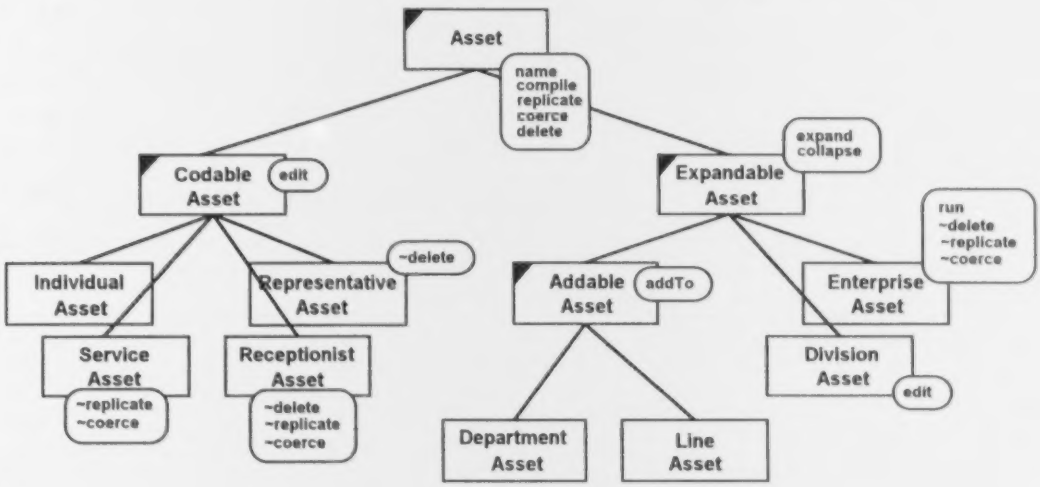


Figure 4.12: The asset inheritance tree

ReplicableAsset and DeletableAsset were merged with Asset. The rounded rectangles contain the main messages defined by each class and the symbol ~ means that a message was overridden because it should not exist for a class. For example, the ReceptionistAsset class overrides the replicate, coerce, and delete methods. The Division class was made a subclass of ExpandableAsset instead of IndividualAsset. The code editing methods were then re-implemented in DivisionAsset. In addition to these changes, the Asset class was made a subclass of the Smalltalk pre-defined class CompositeView so that all assets could inherit the behavior of visual objects that have sub-parts.

Chapter 5.

Program Animation

Enterprise has a program animation component that can be used to monitor a program's performance and to help identify certain types of parallel programming and logic errors. The user can examine the amount of parallelism, when and where synchronization occurs, which machines are being used and their loads, the lengths of message queues, and the state of each process during execution. Currently, there are no debugging facilities for setting breakpoints or examining the values of variables. Animation consists of displaying asset states, displaying messages as they move between assets, and displaying message queues.

This is similar to the mona console used in the JADE system [JLU87]. Both display a view of the state of the program at a point in time, rather than event histories like the JADE text consoles or the time-process views used in HeNCE [BGD91], PIE [SR89], or Taylor's Hermes debugger [Tay92].

5.1. The Animation Model

When an Enterprise program runs, events are captured and logged to an *event file*. The animation system then replays the run using the file. The options dialog box for the program contains a set of radio buttons used to specify whether or not events are captured. If event logging is turned on when a program is executed, the program can be animated after the run by selecting Animate from the design menu. The design view described in Chapter 4 will then be replaced by an animation view, which is described below.

At run-time, Enterprise assets become processes which communicate with each other by sending messages. Assets in the interface maintain a state. Animation proceeds in a series of steps, each step corresponding to one event. Events cause messages to move between assets and assets to change state.

During animation, the time between animation steps is proportional but not equal to the real-time program execution. The proportionality factor can be adjusted by the user to speed up or slow down the animation. The user can also execute the animation one event at a time.

5.1.1. The Animation View

When the user selects Animate from the design view menu, the design view is replaced by an *animation view*. Here, assets have message queues associated with them and all replicas of replicated assets are visible. The state of assets is displayed and messages move from assets into message queues. Figure 5.1 shows a line of 3 assets in the design view. Figure 5.2 shows the same line in the animation view at a point during a replay.

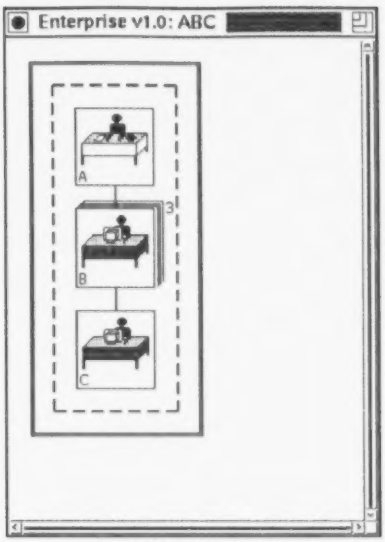


Figure 5.1: A line in the design view

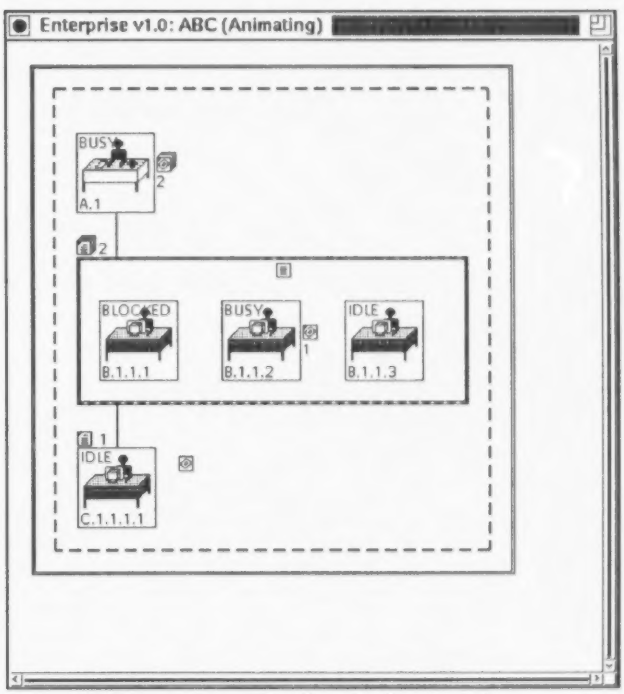


Figure 5.2: The line during a replay

5.1.1. Message Queues and Messages

When an asset call is made, a *call message* containing the parameters moves from the caller to the receiver. For function calls or procedure calls that change their arguments, a *reply message* containing the return values moves from the receiver back to

the caller when the job is done. Messages are represented by small rectangular icons. Different icons are used for calls and replies. This helps the user distinguish between them in complicated programs with several messages moving concurrently. In Figure 5.2, a message can be seen above B.1.1.2 on its way to B.1.1.3. A reply can be seen beside C.1.1.1.1 on its way to B.1.1.1.

Assets have icons that represent queues for incoming calls and received replies. Figure 5.3 shows an asset with input and reply message queues. The queue for incoming calls is called the *input queue* and is located just outside the top left corner of the asset. The location corresponds to the fact that assets handle calls by executing from their first program statement. The queue for received replies is called the *reply queue* and is located just outside the center of the asset's right side. The location corresponds to the fact that replies can be received from any point in an asset's code.

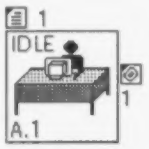


Figure 5.3: An asset with message queues

When a message arrives at its destination, it first moves to just outside an input or reply queue, then into the queue. The queue indicates the number of messages it contains in two ways: it displays a count and it uses different icons when it contains 0, 1, or more messages. The icons for messages, replies, and message queues are shown in Figure 5.4. Queues with 0 messages are not displayed. When animation is stopped, the user can select a message queue and display a menu of the messages it contains. Selecting one of the messages will display its parameters.

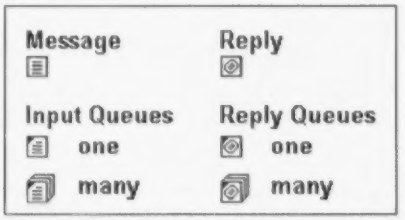


Figure 5.4: Icons for messages and message queues

The messages are removed from the queues in response to events. This is explained in more detail in the events section below.

If a composite asset like a department, line, or division is collapsed in the animation view, it will have an input queue but no reply queue. The input queue is shared with the receptionist for the composite asset. There is no reply queue because the replies are for component assets inside the composite asset, not for the composite asset itself. If the user wants to see replies, the composite asset must be expanded.

Replicas

When a program is being edited in the design view, replication is indicated by drawing lines outside the asset icon and displaying its replication factors. During animation, however, each replica is independent and must maintain its own state. The animation view shows all replicas. The asset's maximum replication factor is used to determine the number of replicas created.

The replicas are displayed inside a rectangle that represents the replicated asset as a whole. The rectangle represents an entity used by the executive called a *manager*. The manager is responsible for allocating work to replicas and keeps track of which replicas are busy and idle. Messages sent to replicated assets actually go to the manager, which then sends them to the appropriate replica. Replies move from the replica directly back to the original sender. Managers have an input queue located just outside of the top left corner of their rectangle, but have no reply queue.

All assets, including managers and replicas, have a name that uniquely identifies them. The name is built by appending a system-generated suffix to the base name assigned by the user. The suffix is generated by the run-time executive based on the communication paths between assets. The executive builds a graph whose nodes are assets and whose arcs represent communication links. The root of the graph is the component of the program's enterprise asset. An asset's children are all assets it can legally call. The following rules are then used to generate suffixes:

- Suffixes consist of digits separated by periods.
- Composite assets share their suffix with their receptionist.
- The root asset is assigned the suffix '.1'
- An asset's parent is defined as its node's parent in a depth-first traversal of the graph.
- Assets build their suffix by appending a '.' and their replica number to their parent's suffix. Replica numbers are integers from 1 to the asset's maximum replication factor.

In Figure 5.2, B is replicated 3 times, so we have a manager and 3 replicas of B. The manager is connected to A.1, so it is named B.1.1. It is represented by the rectangle around the 3 replicas of B, but its name is not displayed. The replicas are all connected to the manager, so they are named B.1.1.1, B.1.1.2, and B.1.1.3. C is connected to all 3 replicas. The leftmost replica has a suffix of .1.1.1, so C is named C.1.1.1.1.

5.1.2. States

As an asset executes, it can be in one of four states: idle, busy, blocked, or dead. An asset changes state in response to events that affect it. The following states are used in the system:

Idle An idle asset is one that is not currently executing. It is waiting to receive a message. The next message sent to it will be received and processed immediately. Initially, all assets are idle.

Busy A busy asset is one that is executing code in response to a message from a caller asset. It can send messages to other assets and receive replies from them, but cannot process a message from another caller until it completes the

active message. Any messages sent to a busy asset will be put into its input message queue.

Blocked A blocked asset is one that has stopped execution to wait for a reply to a message it has sent. This occurs when an asset tries to access the return value from a called asset that has not yet replied.

Dead A dead asset is one that has stopped execution because of some kind of error. The Enterprise executive has determined that it can no longer communicate with that asset.

The state of a collapsed composite asset is defined by the states of its components. If at least one component is busy, the asset is busy. This indicates that the asset is doing some useful work. If no component is busy and at least one is blocked, the composite asset is blocked. This indicates that some component is waiting because of synchronization. If no component is busy or blocked, and at least one is idle, the composite asset is idle. This indicates that no work is being done. If no component is busy, blocked, or idle, they must all be dead. In this case the composite asset is dead.

The state of an asset is indicated by one of two user-selectable mechanisms: color or state name display. If color is used, icons for busy assets are green, idle assets are yellow, blocked assets are red, and dead assets are black. If the state name display is used, the name of the state is displayed at the top left of the asset's icon as shown in Figure 5.2.

5.1.3. Events

Assets change state in response to events that occur when the program is running. An event logging process in the executive monitors programs as they run, identifies when important events occur, and writes event records to the event file. This process is responsible for capturing events, determining the partial ordering between them [Fid88], and assigning a real time to them.

The event file is an ASCII text file. Each event starts on a new line. It begins with a # character followed by an event type and parameters separated by spaces. An optional sequence of information strings can appear on the following lines before the next event. The information strings are displayed when the user inspects message contents. Event parameters depend on event types. They include asset names, message tags, and timestamps. Asset names are the names with suffixes described above. Tags are integers that are used to associate message sends with message receives. The combination of the name of the sending asset and the message tag uniquely identifies a message. Timestamps are integers representing times in milliseconds. They are measured from some arbitrary start time and refer to the time that the event was inserted into the event file. The sequence of times must be monotonically non-decreasing. The animation system modifies the timestamps by subtracting the time of the first event, then multiplying the result by a scaling factor. The scaling factor can be set by the user to adjust the speed of the animation. The event file is described in detail in Appendix B.

The animation system reads the events from the event file and processes them in order. Processing an event can cause assets to change state and messages to move between assets. Seven events are supported: SentMsg, RcvdMsg, Block, SentReply, RcvdReply, DoneMsg and Die.

Figure 5.5 is a state-transition diagram that summarizes the relationship between the asset states, represented by circles, and the events, represented by arrows.

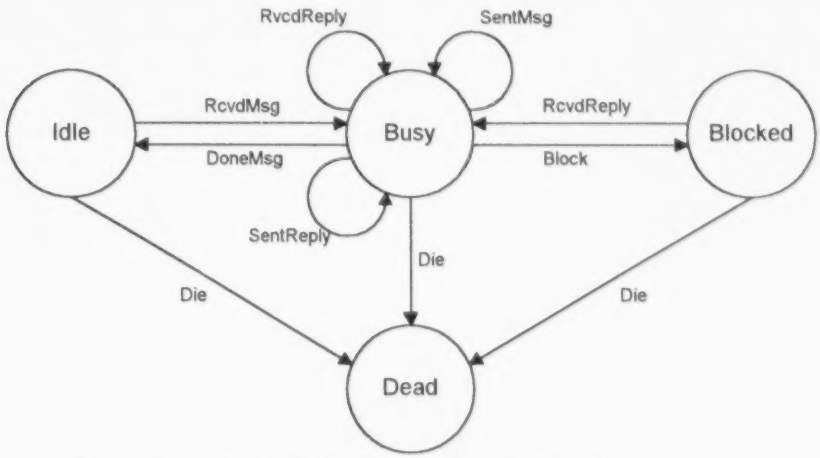


Figure 5.5: The state transition diagram for enterprise assets

SentMsg When the event logging process detects that an asset has sent a message to another asset, it inserts a SentMsg event in the event file. The information strings contain the names and values of all message arguments. During animation, a message moves from the sender to the input queue of the receiver. The sender must be busy and it does not change state. The receiver does not change state.

RcvdMsg When the event logging process detects that an asset has received a message and started processing the task that the message represents, it inserts a RcvdMsg event in the event file. The receiver must be idle and the message must be in its input queue. During animation, the receiver removes the message from its input queue and changes its state from idle to busy.

Block When the event logging process detects that an asset has tried to access a result computed by another asset, and the result is not available, it inserts a Block event into the event file. The asset must be busy. During animation, the asset state changes from busy to blocked. When the result becomes available and the asset resumes executing, the event logging process inserts a RcvdReply event in the event file.

SentReply When the event logging process detects that an asset has sent a reply message to its caller, it inserts a SentReply event into the event file. The information strings contain the names and values of all message arguments. During animation, a message moves from the sender to the reply queue of the receiver. The sender must be busy and does not change its state. The receiver does not change state.

RcvdReply When the event logging process detects that an asset has accessed a return value in a message reply, it inserts a RcvdReply event into the event file. The

asset may be busy or blocked. The reply must be in the asset's reply queue. During animation, the message is removed from the reply queue. If the asset is busy, it remains busy. This situation occurs when a call is completed before the return value is needed. If the asset is blocked, it becomes busy. This situation occurs when an asset was blocked waiting for the value it has just received in the reply.

DoneMsg When the event logging process detects that an asset has finished executing a message, it inserts a DoneMsg event in the event file. The receiver must be busy. During animation, the receiver changes its state from busy to idle.

Die If the event logging process determines that an asset is not responding for some reason, it inserts a Die event into the event file. During animation, the asset becomes dead, but the message queues are not affected. The asset can be in any state before this event.

Note that when a rcvdMsg event occurs it means that the code for an asset has started executing, not that a message transmitted across the network has arrived. This matches the user model of assets interacting and avoids introducing implementation details. When an asset makes a call to another asset, the code inserted by the Enterprise pre-compiler builds a data structure containing the call parameters and transmits it across the network from the machine running the calling asset to the machine running the called asset. The called asset's code is invoked by a process created by the run-time executive that receives the transmitted data and calls the asset. This process consists of a loop that receives a message then calls the asset. If the asset is busy executing a previous call when a new message arrives, the message is stored in a buffer until the call completes and the next iteration of the loop occurs. To hide these details from the user, the animation system indicates that the call has been made by the caller by generating a sentMsg event and indicates that the caller's code has started executing by generating a rcvdMsg event.

Similarly, when a rcvdReply event occurs it means that an asset has consumed a result of a call it previously made, not that a message containing the result has arrived from the called asset. This way, the order of block and rcvdReply events indicates how lazy synchronization affects the parallelism acheived by a program. If a result of a call is accessed before it has been computed and transmitted back to the caller, the caller stops executing and a block event is generated. When the result arrives and the caller resumes execution, a rcvdReply event is generated. If a result has been returned before it is accessed, no block event is generated.

5.2. Using the Animation System

This section describes the user interface of the animation system. Like editing, animation is controlled from menus associated with assets and with the program as a whole. Clicking on an asset displays the asset menu, and clicking outside of any asset displays the animation menu. Clicking on a message queue displays a menu of messages it contains. A message can then be selected and its parameters can be examined.

5.2.1. The Animation Menus

As described in Chapter 4, when the user selects Animate from the design view menu, the design view is replaced by the animation view where all replicas are displayed, messages and message queues appear, and assets are assigned unique names. As in the design view, clicking the middle mouse button when the cursor is outside all assets will display a menu of global operations. In the animation view however, clicking on an asset will display an asset menu only if animation is stopped. In addition, clicking on a message queue will display a message queue menu.

The Animation Menu

Clicking outside all assets and message queues displays the *animation menu*. Its contents change, depending on the current state of the system. When an animation is running, the menu contains the single choice Stop. Selecting Stop will stop the animation after the current event has been processed. All messages will move into their destination queues before the system stops.

When the system is stopped, the animation menu contains: either the single choice Start or both Reset and Resume, Step, Set Speed, either Color On or Color Off, and Design.

Start This choice will only appear if the first event in the animation has not been processed. Otherwise the choices Reset and Resume will appear. Selecting Start will cause the system to begin processing events and will change the menu to contain the single choice Stop.

Reset This choice will only appear if an animation has been stopped after one or more events have been processed. Selecting Reset will reset the system to its initial state so that the animation can be restarted from the first event. The menu choices Reset and Resume will be replaced with Start. Animation will not be started until Start is chosen from the menu.

Resume Like Reset, this choice will only appear if one or more events have been processed. Selecting Resume will cause the system to continue processing events from where it was stopped. The menu will change to contain the single choice Stop.

Step This choice will only appear if there are more events to be processed. Selecting Step will process one event, then stop.

Set Speed Selecting Set Speed will display a dialog box asking the user to enter a scaling factor. Typing a number into the box, then pressing ENTER will set the system's scaling factor for event timestamps. Pressing ENTER with nothing in the box will leave the current scaling factor unchanged. The scaling factor is used to adjust the speed of the animation so that events can be replayed in time proportional to real time without the system falling behind. The algorithm used and the effect of the scaling factor are described in the implementation section below.

Color On This choice will only appear if asset states are currently being displayed using state names. Selecting Color On will switch to using colors instead. The choice Color On in the menu will be replaced with Color Off.

Color Off This choice will only appear if asset states are currently being displayed using color. Selecting Color Off will switch to using state names instead. The choice Color Off in the menu will be replaced with Color On.

Design Selecting Design will switch from the animation view back to the design view. The animation view can be re-displayed by selecting Animate from the design menu.

Asset Menus

Clicking on an asset when animation is stopped displays an asset menu. Because no editing of the asset hierarchy or code is allowed, the only choices that appear in animation view asset menus are Expand and Collapse. They only appear for departments, lines, and divisions, and perform the same function they do in the design view. Other assets do not currently have an asset menu in the animation view.

Message Queue Menus and Messages

Clicking on a message queue when animation is stopped displays a menu of messages that the queue contains. The menu displays the name of the sender and the message tag for every message in the queue. Selecting a message from the menu will display a window containing the information strings for the event that generated the message. Currently, the information strings contain message arguments. They are built by the run-time executive and passed to the animation system for display. For example, if asset A.1 calls B.1.1, passing it two arguments a and b with values 3 and 4 respectively, the event file will contain the event:

```
#sentMsg A.1 B.1.1 1 1000
a=3
b=4
```

When the event is processed, a message will be created that stores the information strings 'a=3' and 'b=4'. The message will animate from A.1 to B.1.1 and into B.1.1's input queue. If animation is then stopped, B.1.1's input queue menu will contain the choice: 'from:A.1 tag:1'. Selecting that choice will display a window containing the strings 'a=3' and 'b=4'.

The information strings could also be used to pass other run-time information like the machine CPU load or memory usage.

5.3. Implementation

This section discusses the implementation of the animation system. Several new classes were added to the user-interface to support animation and several behaviors were added to the existing classes.

5.3.1. The Animation View

When the animation view is displayed, the asset graph is modified. Each replicated asset is wrapped in an instance of ReplicatedAsset that contains the original asset together with a list of replicas that are constructed by copying the original asset. The copies are identical, except that each is given a different replica number. As an animation proceeds, the states of these replicas may diverge. The ReplicatedAsset

represents the manager of the replicas. It is also responsible for drawing the rectangle around the replicas, much like ExpandableAssets do around their components.

Two new responsibilities are added in the Asset class: knowing the input message queue and knowing the reply message queue. Both queues are instances of the subclasses of MessageQueue, which are InputQueue and ReplyQueue. A MessageQueue contains an ordered collection of messages, which are instances of class Message. The display method in Asset checks to see if animation is active and if so, allocates room for the message queues when it computes its bounding rectangle. When an asset is told to draw itself, it also tells its message queues to draw themselves.

Message queue selection is implemented by augmenting the message that is sent to an asset to ask it for its sub-asset that contains the cursor point. An asset now considers its two queues as candidates in addition to its component assets. A MessageQueue instance determines if it contains the cursor point by testing if the point is within its screen extent. Thus the method that determines the asset at a point may now return a message queue or an asset. In either case the object is asked for its menu by sending it the 'menu' message. MessageQueue's menu method simply builds a menu from the names of the messages it contains.

5.3.2. The Animation Architecture

The animation architecture has an asynchronous component that is responsible for processing events at the correct animation time and a synchronous component that is responsible for animating messages. Both components periodically stop to allow the system to check for user input.

The Control Model

Two new objects were added, EventQueue and AnimationQueue. An EventQueue contains a collection of events read from the event file. It processes events asychronously. An AnimationQueue contains a collection of objects that are to be moved on the screen in discrete steps. It animates messages synchronously. An enterprise contains an instance of EventQueue and an instance of AnimationQueue. When animation is active, the control loop for the window sends the message 'animate' to the enterprise every time through the loop. The enterprise responds by telling its animation queue to animate its objects and telling its event queue to process its events. The animation queue moves each of its objects one step along their path, then returns. The event queue processes its events in order until the event time equals the current time. Control is then returned to the control loop, which checks for user input. In this way the animation system only takes control periodically and, when it does, only for a short time.

The enterprise maintains a flag that is false if animation is stopped, true when animation should be running. The animate method checks the flag and only processes the event and animation queues if the flag is true. When the user first switches to the animation view, the flag is set to false so that animation is stopped. When Start or Resume is selected from the animation menu, the flag is set to true, causing animate to begin processing events and animating messages. If the user then selects Stop from the menu, the flag is set to false causing event processing and message animation to stop.

One problem had to be solved before the control model worked properly. As described in Chapter 4, Smalltalk-80 polls the controllers of all active windows to

determine if any want control. In addition to checking to see if their views have the cursor, controllers also share a semaphore. If the semaphore is set, the controllers block until it is cleared. The semaphore is set if the user does not interact with the system periodically. Thus the animation would stop executing unless the user kept moving the mouse every few seconds. The problem was partially solved by overriding the Enterprise controller's control loop so that it did not check the semaphore. However, the animation will still stop if the cursor is in another window.

The Asynchronous Component

An instance of EventQueue is responsible for knowing the start time for an animation and the events from an event file. It is created when the animation view is first displayed. That is, to speed up event processing, the event file is parsed and all events are created before the animation begins. The animation start time is set when the user actually starts an animation.

The events in an event queue are instances of subclasses of AnimationEvent. When the event file is parsed and animation events are created, the timestamps in the event file are translated to times relative to the start time for the event queue before storing them in the events. The first event is assigned time 0. All other event times are calculated by subtracting the original timestamp for the first event from the timestamp for the event.

The event queue maintains a start time and a pointer to a current event. When the enterprise tells the event queue to process events, the event queue looks at the time stored in the current event. If the current time of day is later than or the same as the event time plus the start time for the event queue, the event is told to process itself and the pointer is moved to the next event. This continues until the current time is earlier than the event time plus the start time or we reach the end of the events. The event queue then stops processing events and returns control to the enterprise. To allow the user to interact with the system, it must process events fast enough so that it can return control to the enterprise frequently. To ensure this, the event times are actually multiplied by a scale factor before being added to the start time. That is, the condition for processing an event is that

```
current_time >= (start_time + scale*event_time)
```

where `current_time` is the current time of day, `start_time` is the start time stored in the event queue, `scale` is the scale factor, and `event_time` is the time stored in the event. The scale factor can be set by the user by selecting Set Speed from the animation menu. A larger scale factor will increase the real time between events, allowing the event queue to process fewer events each time through the control loop, which lets the animation system animate messages and check for user input more often.

The event queue sets its start time when it is asked to reset or to resume. When asked to reset itself, the event queue sets its current event pointer to its first event and sets its start time to the current time of day. It is then ready to begin processing events starting at its first event. When asked to resume processing events, the event queue does not change its current event pointer, but sets its start time to the current time minus the scale factor times the time in the current event. It is then ready to begin processing events starting from the current event.

Each animation event represents one event from the event file. In addition to the event time, an animation event contains a collection of responses. Each response consists

of an asset, a message selector, and an array of arguments for the message. When a response is processed, the message specified by the selector is sent to the asset using the arguments. One event may contain several responses. For example, a SentReply event contains two responses: one to tell the sending asset it has sent a reply and one to tell the receiving asset it has been sent a reply. The set of responses for one event is treated as a transaction; if one message is sent they all are. There is a subclass of AnimationEvent for each type of event. Each subclass only implements creation methods. All other behavior is implemented in AnimationEvent.

In addition, the asset classes implement methods for the messages sent by responses. For example, one of the responses for a SentMsg event tells the receiver that is has been sent a message by the sender by sending a Smalltalk message to the receiver, passing it the sending asset, the message tag, the event time, and an array of information strings. The receiver will be one of the Asset subclasses. Asset implements the method for the response. The method creates an Enterprise call message, determines the path it must follow to move from the sender to the receiver's input queue, and puts the message into the animation queue. The message will then be moved along the path by the animation queue as described below.

The Synchronous Component

An instance of AnimationQueue is responsible for maintaining a collection of objects to be animated and for moving them along pre-computed paths. Initially, the enterprise's animation queue has no objects to animate. As the event queue processes events, it creates Enterprise call and reply messages and inserts them into the animation queue. Along with other information, each message knows the path it must follow.

Animation proceeds in a sequence of equally spaced time steps. This *step time* is stored in a class variable of AnimationQueue. When the enterprise tells its animation queue to animate its objects, the queue checks to see how much time has elapsed since it last animated. If the elapsed time is longer or the same as the step time, it tells each of its objects to animate itself. Each object responds by moving itself to the next point on its path, then returning true or false. Any object that returns false is removed from the animation queue. The animation queue then returns control to the enterprise. To ensure that animation is smooth, the animation queue must be told to animate frequently enough that the elapsed time will be close to the step time.

Messages move from the sending asset to either the input queue or the reply queue of the receiving asset. Because the destination queue is part of the receiver, the message is actually created by the receiver. When a SentMsg or a SentReply event is processed by the event queue, one of the responses informs the receiver that it has been sent a message, passing it the sending asset, message tag, event time, and information strings as arguments. The receiver then determines the path a message must follow to move from the sender to the destination queue, creates a message of the appropriate type, and adds the message to the animation queue. The first time it is told to animate, the message draws itself at the first point on its path, deletes the point from the path, and returns true. Each time the message is then told to animate itself, it erases itself from the display, draws itself at first point on its path, deletes the point from the path, and returns true. If the path is empty, the message erases itself from the display, tells the destination queue that it has arrived, then returns false so that it will be removed from the animation queue.

The paths are created in three steps. First, the sending asset is asked for the point where the path should start, and the destination message queue is asked for the point where the path should end. Using these, the corners of the path are determined. Two corners are generated for call messages, three for replies. Once the corners are known, evenly spaced points are found along the lines from the start point, to each corner in turn, ending at the end point. Figure 5.5 shows the paths generated. The crosses mark the start and end points, and the bullets mark the corners.

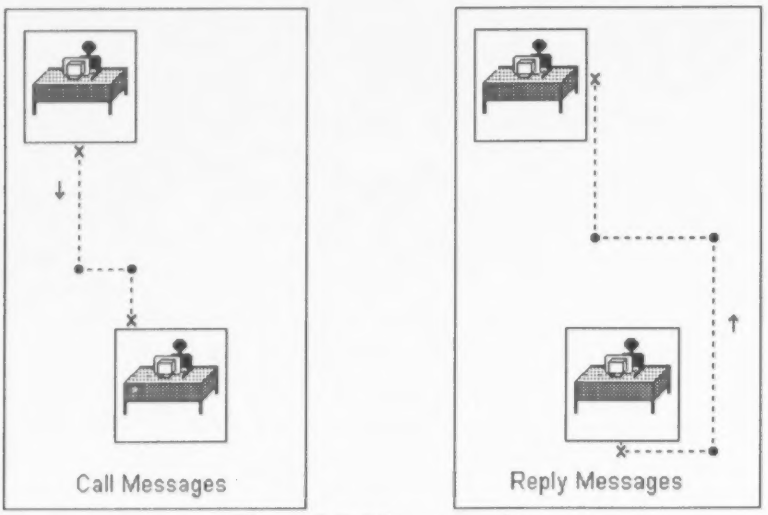


Figure 5.5: Message paths

A problem arises because of the interaction between the asynchronous event processing and the synchronous message animation. When a RcvdMsg or a RcvdReply event is processed, the system assumes that the message has reached the receiver's input or reply queue. The message is removed from the receiver's queue and the receiver changes its state to busy. However, there are cases where a message may not have animated all the way to its destination queue when the RcvdMsg or RcvdReply is processed. In this case the receiver would change to busy before the message reached it. To fix the problem, messages have a state that can be either pending or received. When a message is put into the animation queue, it is also put into the destination message queue, but marked as pending. Message queues do not include pending messages in the count of messages they display to the user. When a message reaches its destination, it marks itself as received, causing the destination queue to include the message in its count. When a RcvdMsg or RcvdReply is processed, the receiver first determines the status of the message. If the message is pending, it is removed from the animation queue and erased from the display before it is removed from the receiver's message queue or the receiver changes state. The result is that a message that has not reached its destination will disappear from the display, then the asset will change to busy. This synchronizes the events presented to the user without slowing down the animation system.

Chapter 6.

Conclusions and Future Work

6.1. Unimplemented Features

Because of time constraints, some of the non-essential features described in the thesis were not implemented:

Design View

- The .ENTrc file has not been implemented. Instead, the EDITOR option and the ENTERPRISE path are specified using UNIX environment variables.
- Deleting a program has not been implemented. A program can be deleted by deleting its directory tree from outside the interface.
- Renaming an enterprise has not been implemented. An enterprise can be renamed from outside the interface by renaming its directory, then renaming the graph and event files in its Graph sub-directory.
- Renaming an asset has not been implemented. An asset can be renamed from outside the interface by renaming its source code file in the program's Assets sub-directory, then changing the asset's name in the graph file.
- Compile All has not been implemented. The system can be made to re-compile all assets by changing their time of last modification from outside the interface using the UNIX `touch` command.
- Compiling assets from asset menus without linking has not been implemented. The Enterprise compiler can be invoked from outside the interface to compile individual assets if needed.
- The libraries from the program compile options dialog box are not passed to the linker. For programs that use libraries, the linker must be invoked from outside the interface.
- The verbose and event logging flags are always on. The values set in the dialog box are remembered and stored in the interface, but they are not passed to the run-time executive.
- When a composite asset is re-classified as an individual, the code from its components is not concatenated into one file. The user must do it manually from a code editor. The files for the components are not deleted, but will still exist in the Assets sub-directory.
- If Execute is chosen after a program's graph has been changed, no check is made to ensure that the program was re-compiled. This could result in a program whose graph file does not match its executable code.

Animation View

- There are no asset menus, message queue menus, or message windows.

- Composite assets must be fully expanded before animating the program. If they are left collapsed, the system still assumes they are expanded and messages will move to the wrong places on the display.

6.2. Problems with the Current Implementation

During implementation, a few problems became evident:

Program and Asset Options

The options set in the asset options and program options dialog boxes are not saved between sessions. They are re-set to their default values every time the program is re-loaded for editing. It would be better to save and restore the options set by the user. This could be done by adding a file to hold program options. The asset options could be saved by adding sections to the graph file that would be used only by the interface.

Pending Messages

It is confusing when pending messages are removed from the display by a RcvdMsg event before they reach their destination queue. The receiving asset changes its state to busy, but it is not always clear where the message was going. Instead of removing a pending message from the display, the RcvdMsg event could be forced to wait until it arrived. This could be implemented by having events return either true or false when the event queue tells them to process themselves. True would mean that the event was processed successfully. False would mean that the event was not processed and should be re-tried later. When an event returned false, the event queue would not increment its current event pointer, but would stop processing events and return to the control loop instead. The next time it processed events, the event queue would start with the same event. To ensure that they eventually get processed, events that return false should eventually return true. Another solution would be to have the event queue limit the number of re-tries.

Message Speed

Currently, when message paths are created, every path contains the same number of points. The distance between points is calculated based on the length of the path through the corners. This makes each message take the same time to move from its sending asset to its destination queue. Replies usually have longer paths, so they move faster during animation, giving the false impression that the actual message transmission was quicker. Also, long paths may have points far enough apart that the animation becomes jumpy.

It would be better to have all messages move at the same speed. Although some messages would take longer to move than others, it would be less misleading than moving them at different speeds and the animation would always be smooth. This could be done by using the same distance between points on every message path.

Asset Suffixes

The algorithm for generating asset suffixes builds long asset names that are difficult to read and display. The purpose of the suffix is to ensure that every asset in a program has a unique name at run-time, but the names were not designed to be readable by human beings.

It would be better if the interface used a more readable naming scheme for display and converted between the two suffixes when communicating with the executive. The names could then be made more readable and the suffixes could be based on the structure of the graph instead of the structure of the communication links.

Event Files

For real programs, the event files may be large. It is impractical to read and parse the entire file before starting animation. It takes too long and it may require too much memory to store the entire event queue. Instead, the event file could be read in blocks. Whenever there are no more events in its collection, the event queue can read another block from the file.

Currently the event files consist of ASCII characters. This made them easier to create during development and testing, but makes them too large for real programs. Events should be encoded into binary files instead. Using binary files would make them much smaller. For example, a 4 bit field could represent any of the seven event types and still leave room for expansion, whereas it currently requires an average of 9 bytes.

Animation

When the animation system was implemented, several problems became evident. Initially, messages were moved by asking their wrappers to move. This was simple to implement because wrappers already knew how to move themselves without damaging the background. The strategy failed, however, when several messages were being moved at once. When a wrapper moves itself it invalidates the rectangle at its previous position, causing the view it was contained in to re-draw the background, then it draws itself at its new position. The result was that the entire Enterprise window was told to re-draw itself once for every point on the path of every message. Even though the re-drawing was confined to small damage rectangles, the animation was very slow.

To solve the problem, the system now handles restoring the background itself. Before drawing a message, the background is saved. At the next step of the animation, the saved background is restored, the background at the new location is saved, and the message is drawn at its new location. Smalltalk provides a built-in method that implements this. This made the animation faster and smoother, but caused another problem. The built-in method moves one object along its entire path each time it is invoked. The result is that the first message moves along its entire path, then the next one moves along its entire path, and so on, until the entire animation queue has been processed. This prevents the animation queue from returning control to the control loop until all animation is done, disrupting the timing of subsequent event processing.

The system method should be re-implemented so that it will move each message one step along its path, then return control like the original algorithm did. It may be difficult to find an efficient algorithm to do this, however, because the source and destination rectangles of different messages may overlap.

6.3. Future Work

Current Work

Enterprise is an ongoing project, and several things are currently being done to improve the interface and animation components:

- The missing features in the interface and animation components are being implemented.
- Work is being done to implement input and reply queues for collapsed composite assets. The input queues will be shared with the receptionist. The reply queues will contain only those replies that come from assets outside the collapsed asset.
- The states of collapsed composite assets are being re-defined. The current model is misleading and is inconsistent with the states of the components. For example, a busy or blocked composite asset's receptionist could be idle and thus still receive messages. During animation, the blocked asset could consume messages from its input queue.
- Currently, the manager for replicated assets cannot be collapsed, does not have a state, and has no reply queue. Work is being done to define and implement these.
- The synchronous component of the animation architecture is being re-designed to solve the problems described in the previous section.

Future Enhancements

Several ideas have been put forward for enhancements:

- It would be useful to have performance meters or memory usage statistics displayed for the machines running specific assets. This could be implemented by adding a switch to the run-time options that causes the run-time executive to capture the relevant data and log it to the event file. The asset menus in the animation view could then have choices that enable the data displays during animation.
- Information about messages could be encoded in the way they look or move. For example, messages that contain a large amount of data could be drawn with bulging sides, or messages could be traced by having them leave colored trails behind them.
- The interface could support multiple graph files for the same program. This would allow using the same source code in different configurations. It would also be useful to have several event files for a given graph file. The user could then compare different runs without re-executing a program just to generate an event file.
- An event browser could be provided that allows the user to view the entire event file and perform operations on events. Breakpoints could be set this way.
- Machines could be specified by type or selected features. For example, the user may want all Sun-4 machines that have more than 8 megabytes of memory. A database of available machines could be maintained and made accessible to the user. In addition, machines could be assigned access permissions so their owners could prevent their use at certain times or under certain conditions.

Real-Time Animation

An eventual goal of the animation system is to provide real-time animation of a distributed program. This presents some problems for the current system that must be solved:

- Event logging is now done by one process in the executive. It implements an algorithm to determine the partial ordering between events. It also assigns events

a timestamp by reading its time of day clock when it writes the event to the event file, then sorts the event file by timestamps when the program ends. In real time, the algorithm for assigning real times would have to be more sophisticated because there would be no opportunity to sort the file. These problems have been addressed in other work [Tay92], and similar solutions will have to be implemented.

- The animation system may not be able to process events fast enough to keep up in real-time. Even if it could, messages would move between assets in fractions of second, too fast for the user to see them. The purpose of showing messages moving is to indicate that communication across the network has occurred. This could be indicated instead using methods like those used in JADE [JLU87] that do not require objects to move on the display.

6.4. Conclusions

Using object-oriented techniques and Smalltalk made implementing the interface much easier than using other software design methods or languages. Two attempts to implement the interface using C and X Windows tool kits failed because they required more time and resources than the project could provide. Using Smalltalk, a working prototype was built in three weeks. The system was then developed as an evolving prototype, which worked well in conjunction with the object-oriented design and programming models. The design had to be flexible because the specifications changed at almost every project meeting as features were rejected or modified after seeing them in the prototype. Adding or modifying features was easy, and did not require large changes to existing code.

The animation architecture is application independent. The event queue consists of event objects which must understand the 'process' message and know the time at which they occur. Any object that meets these requirements can be an event. Processing the events can then be done by periodically telling the event queue to process itself. Similarly, the animation queue can animate any object that understands the 'animate' message. The object must respond to the message by performing one step of its animation. For example, assets could implement an animate method that switches between two different icons. They could then be made to flash by adding them to the animation queue.

We have a small user community that has used Enterprise to write several programs. It has proven to be easy to map applications to Enterprise assets [Par93]. The user interface is intuitive to use because operations are always accessed from one of two menus, and only legal operations are ever available. It is easy to experiment with different types of parallelism in an application by simply modifying the graph and re-compiling the program. In most cases the code does not need to be changed at all. The animation system has been useful for identifying performance bottlenecks and wasted resources. Presenting the execution graphically summarizes large amounts of data in an easily interpreted form. There is no need to insert code to gather statistics or to analyze them after a run. Even though the animation system does not currently support real-time animation, it is useful in its present form and will continue to be provided in future versions of Enterprise.

References

[BDG91] A. Beguelin, J.J. Dongarra, G.A. Geist, R. Manchek, V.S. Sunderam. Graphical Development Tools for Network-Based Concurrent Supercomputing. *ACM Supercomputing*, pp. 435-444, June 1991.

[Cha92] E. Chan. The Enterprise Code Librarian. M.Sc. thesis, Dept. of Computing Science, University of Alberta, 1992.

[Fid88] C. J. Fidge. Partial Orders for Parallel Debugging. *Proceedings of the 1988 Workshop on Parallel and Distributed Debugging*, ACM, pp. 1-10, 1988.

[ISIS92] ISIS Distributed Systems, Inc., Ithica, N.Y. *The Distributed ISIS Toolkit: Version 3.0 User Reference Manual*, ISIS Distributed Systems, 1992.

[JLU87] J. Joyce, G. Lomow, and B. Unger. Monitoring Distributed Systems. *ACM Transactions on Computer Systems*, Vol. 5, No. 2, pp. 121-150, May 1987.

[JS80] A. Jones and A. Schwartz. Experience using Multiprocessor Systems - A Status Report. , *Computing Surveys*, Vol. 12, No. 3, pp. 121-166, 1980.

[Lam78] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *CACM*, Vol. 21, No. 7, pp. 558-565, 1978.

[LLM92] G. Lobe, P. Lu, S. Melax, I. Parsons, J. Schaeffer, C. Smith and D. Szafron. The Enterprise Model for Developing Distributed Applications. *Technical Report TR 92-20*, Dept. of Computing Science, University of Alberta, 1992.

[LP91] W. LaLonde and J. Pugh. *Inside Smalltalk Volume II*, Prentice-Hall, Englewood Cliffs N.J., 1991.

[LSS93] G. Lobe, D. Szafron, and J. Schaeffer. Program Design and Animation in the Enterprise Parallel Programming Environment. *Technical Report TR 93-04*, Dept. of Computing Science, University of Alberta, 1993.

[LSV89] T. Lehr, Z. Segall, D. Vrasalovic, E. Caplan, A. Chung, and C. Fineman. Visualizing Performance Debugging. *IEEE Computer*, pp. 38-51, October 1989.

[LSW87] D. Lanovaz, D. Szafron and B. Wilkerson. The Synergism of Logic-Based Programming and Software Engineering: A Programming Environment Approach. *CIPS Edmonton '87 Intelligence Integration Conference Proceedings*, pp. 43-53, November 1987.

[MH89] C.E. McDowell and D.P. Helmbold. Debugging Concurrent Programs. *ACM Computing Surveys*, Vol. 21, No. 4, 1989.

[NMP88] T.A. Marsland, T. Breitkreutz, and S. Sutphen. NMP - A Network Multiprocessor. *Technical Report TR-88-22*, Dept. of Computing Science, University of Alberta, 1988.

[Par93] I. Parsons. An Appraisal of the Enterprise Model. M.Sc. thesis, Dept. of Computing Science, University of Alberta, 1992.

[PDT93] J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons. The Enterprise Model for developing Distributed Applications. *IEEE Parallel and Distributed Technology Systems and Applications*, 1993, to appear.

[PP90] ParcPlace Systems, Inc. *Objectworks\Smalltalk Release 4 User's Guide*, ParcPlace Systems Inc., 1990.

[SR85] Z. Segall and L. Rudolph. PIE: A Programming and Instrumentation Environment for Parallel Processing. *IEEE Software*, pp. 22-37, Nov. 1985.

[SSG91] A. Singh, J. Schaeffer, and M. Green. A Template-Based Approach to the Generation of Distributed Applications Using a Network of Workstations. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 1, pp. 52-67, 1991.

[SSW92] D. Szafron, J. Schaeffer, P.S. Wong, E. Chan, P. Lu, and C. Smith. The Enterprise Distributed Programming Model. *Programming Environments for Parallel Computing*, N. Topham, R. Ibbett and T. Bemmerl, editors, Elsevier Science Publishers, pp. 67-76, 1992.

[Tay92] D. Taylor. A Prototype Debugger for Hermes. *Cascon '92*, IBM Canada Ltd, Toronto, pp. 29 - 42, November 1992.

[TOOLS93] G. Lobe, D. Szafron, and J, Schaeffer. The Object-Oriented Components of the Enterprise Parallel Programming Environment. *Proceedings of the TOOLS 11 Conference*, pp. 215-229, 1993.

[Won92] P.S. Wong. The Enterprise Executive. M.Sc. thesis, Dept. of Computing Science, University of Alberta, 1992.

[WWW90] R. Wirfs-Brock, B. Wilkerson and L. Wiener. *Designing Object-Oriented Software*, Prentice Hall, 1990.

Appendix A.

The Enterprise Graph File Format

This appendix describes the format of the Enterprise graph file using extended BNF notation.

A.1. Notation

```
<abcd>*  means 0 or more occurrences of <abcd>
<abcd>+  means 1 or more occurrences of <abcd>
```

A.2. Syntax

```
<graph>         ::= <asset>
                    <service>*

<asset>         ::= <name> <simpType> <min> <max> <order> <debug> <opt>
                    <options>
                |   <name> <compType> <min> <max> <order> <debug>
<opt><count>
                    <options>
                    <asset>+

<service>       ::= <name> service <debug> <opt>
                    <options>

<name>          ::= <string>

<min>           ::= <positive integer>

<max>           ::= <non-negative integer>

<order>         ::= ORDERED | UNORDERED

<debug>         ::= DEBUG | NDEBUG

<opt>           ::= OPTIMIZE | NOPTIMIZE

<simpType>      ::= individual | representative

<compType>      ::= line | department | division

<count>         ::= <positive integer>

<options>       ::= CFLAGS <flags>
                    EXTERNAL <libraryList>
                    INCLUDE <machineList>
                    EXCLUDE <machineList>

<libraryList>   ::= <string>

<machineList>   ::= <string>

<flags>         ::= <string>
```

A.3. Semantics

`<graph>`

A graph represents the entire Enterprise program. It consists of an asset definition followed by 0 or more service definitions. The file can be parsed from top to bottom to perform a depth-first traversal of the graph.

`<asset>`

An asset can either be simple or composite. Simple assets are either individuals or representatives. Each is represented by one line containing information about the asset followed by four lines containing information about options. Composite assets are represented in the same way as simple assets except that they also specify a count of children and are followed by a definition for each child.

`<service>`

A service asset is represented in the same way as a simple asset, except that it cannot have a replication factor or ordering option.

`<name>`

A name may be used as the base name of an asset or a C source file.

`<min> and <max>`

These are integers representing the minimum and maximum replication factors. If they are both 1, there is no replication. Min must be > 0 and max must be 0 or >= min. An asset will be replicated at least min times and at most max times. If max is 0, there is no fixed maximum and the asset is replicated as many times as necessary to use all available processors. If max = min, an asset will be replicated exactly min times.

`<order>`

This flag indicates whether a replicated asset's return values are returned in the order that the assets were called (ORDERED) or in the order that they finish (UNORDERED).

`<debug>`

This flag indicates whether an asset should be compiled using debug flags (DEBUG) or not (NDEBUG). It may also be used to turn the debugger on and off for each asset.

`<opt>`

This flag indicates whether an asset should be compiled with optimization off (NOPTIMIZE) or on (OPTIMIZE).

`<simpType>`

The type of a simple asset must be individual or representative.

<compType>

The type of a composite asset must be line, department or division.

<childcount>

This integer is a count of components in the composite asset. It includes the receptionist.

<options>

Four lines give options for compiling, linking and executing each asset and all four lines must appear. If an option does not apply to an asset, the rest of the line is left blank. The options are treated as character strings by the interface. That is, they will not be parsed but will be passed to the Enterprise executive in the form that they are entered by the user. CFLAGS gives a list of compile flags to use when compiling the asset. They are appended to the compile command by the executive. EXTERNAL gives a list of external modules or libraries to be linked with an asset. They are appended to the link command by the executive. INCLUDE gives a list of machines that can execute an asset. If the list is present, the machines will be used instead of the machines in the Enterprise machine file. EXCLUDE gives a list of machines that are forbidden to execute an asset. These will be excluded from the list in machine file.

Appendix B.

The Enterprise Event File Format

This appendix describes the format of the Enterprise event file using extended BNF notation.

B.1 Notation

```
<abcd>*  means 0 or more of <abcd>
<abcd>+  means 1 or more of <abcd>
a|b      means a or b
()       is used for grouping
```

B.2 Syntax

```
<eventFile>     ::= <event>*

<event>         ::= #(<sentEvent> | <rcvdEvent> | <doneEvent> |
<blockEvent>
                        | <dieEvent>) <evTime> <comment>*

<sentEvent>     ::= (sentMsg | sentReply) <assetName> <assetName>
<msgTag>

<rcvdEvent>     ::= (rcvdMsg | rcvdReply) <assetName> <assetName>
<msgTag>

<doneEvent>     ::= doneMsg <assetName>

<blockEvent>    ::= block <assetName> <msgTag>

<dieEvent>      ::= die <assetName>

<comment>       ::= <oneLineOfFile>

<assetName>     ::= <assetBase> <assetSuffix>+

<msgTag>        ::= <integer>

<evTime>        ::= <integer>

<assetBase>     ::= <string>

<assetSuffix>   ::= <integer>
```

B.3 Semantics

`<eventFile>`

An `<eventFile>` contains all of the events that were captured for one run of the program. It consists of zero or more event records. The file is used to communicate between the run-time executive and the animation system.

`<event>`

An `<event>` represents the occurrence of one run-time event. Because each event may span multiple lines in the file, each must be prefixed with the # character. Events are generated in response to actions taken by the user's program. Each event record contains the time at which the event occurred. The sequence of times must be non-decreasing.

`<sentEvent>`

A `<sentEvent>` can be either a `<sentMsg>` or a `<sentReply>`. A `<sentMsg>` is generated by an asset that has sent a message to another asset. A `<sentReply>` is generated by an asset that has previously received a message from another asset and has just sent a reply for this message. In both types, the record contains the name of the sending asset, the name of the receiving asset, and the tag for the message. Following this line is an optional comment. The comment will be displayed in the message when it is expanded by the user during an animation. Each line of comment will be displayed on a separate line in the expanded message.

`<rcvdEvent>`

A `<rcvdEvent>` can be either a `<rcvdMsg>` or a `<rcvdReply>`. A `<rcvdMsg>` is generated by an asset that has received a message from a caller and started to work on the task. A `<rcvdReply>` is generated by an asset that has accessed a reply from a previous call to another asset. In both types, the record contains the name of the receiving asset, the name of the sending asset, and the message tag. The tag must match the tag of a message (for `<rcvdMsg>`) or reply (for `<rcvdReply>`) that was previously sent.

`<doneEvent>`

A `<doneEvent>` is generated by an asset that has finished a task and become idle. If a reply was sent, the asset must generate a `<sentReply>` event before the `<doneEvent>`. The event record contains the name of the asset.

`<blockEvent>`

A `<blockEvent>` is generated by an asset when it tries to access the returned value of a previously sent message and the reply is not yet available. The event record contains the name of the blocking asset and the tag of the message that was sent and has not yet returned.

<dieEvent>

A <dieEvent> is generated by the run-time executive when it detects that an asset is no longer responding to messages. The event record contains the name of the asset that has died.

<comment>

A <comment> is a string of characters with embedded spaces, ended by an end of line. It will be displayed when its message is expanded by the user during an animation. The animation system will not process the string in any way. The run-time executive is responsible for building the string before writing it to the event file.

<assetName>

An <assetName> is a string that matches the name of one of the assets in the graph, including its suffix. An asset's <assetName> is unique within a program, even when replicas are considered. The <assetName> is built by appending its suffix to the base name assigned by the user.

<msgTag>

A <msgTag> is an integer that uniquely identifies a message from a specific asset. The combination of the sender and message tag uniquely identifies a message in the system. Message tags are used to associate message receives with message sends.

<evTime>

An <evTime> is an integer time measured from some arbitrary start time in milliseconds.

Appendix C.

Installation and Setup

C.1. Installation

The interface requires that the user have a licensed copy of Smalltalk-80 version 4.0 installed on their system. The rest of this section assumes that Smalltalk and X windows have been installed.

The interface is distributed as several files containing the required classes. The files have been collected into one file and then compressed by using the Unix `tar` and `compress` commands. The resulting file is named `interface.tar.Z`. This file must be uncompressed and expanded, then the classes must be installed into a Smalltalk image. Installing the interface involves the following steps:

- Create a new directory to contain the interface. The rest of this section calls this the *Enterprise directory*.
- Copy the file `interface.tar.Z` into the Enterprise directory.
- Make the Enterprise directory the current directory.
- *Uncompress* the file using the command `'uncompress interface.tar.Z'`. This will produce a file named `interface.tar`.
- *Untar* `interface.tar` by using the command `'tar -xvf interface.tar'`. This will create a sub-directory named `Filein` holding the class files.
- Create a new Smalltalk image in the Enterprise directory or copy an existing image into it. The image will consist of two files. The rest of this section assumes that the image files have the default names `st80.im` and `st80.changes`.
- Start Smalltalk.
- When Smalltalk comes up, go to the Launcher and select File List.
- Type Filein/* and press ENTER in the top panel of the file list window that appears.
- Select `enterprise.filein` from the middle panel of the window.
- Wait while the classes are filed in. They will be put into a category called Enterprise in the Browser.
- Arrange the windows, then save the image by selecting Save from the Launcher Special menu.
- Exit Smalltalk by selecting Quit from the Launcher Special menu.
- If desired, `interface.tar`, all of the files in the `Filein` directory, and the `Filein` directory itself can now be deleted.

C.2. Setup

The user must next create an ASCII text file named `.ENTrc` in their home directory. When the interface first starts, it reads this file and sets some global options. Each line of the file contains one option and is of the form `<option> = <value>` where `<option>` is the name of the option and `<value>` is the value it is set to. Currently two options are supported:

- `EDITOR = <editor command>`
  This option does not have to appear in the file. If it appears, `<editor command>` is used as the name of a command to run to invoke a text editor for editing code. The command will have the name of a file appended to it when it is invoked. For example, to use the Unix `vi` editor, `.ENTrc` should contain the line: `EDITOR = vi`. Then, to edit a file named `AssetA.e` the interface will execute the command `'vi AssetA.e'`. The `vi` editor will be run in an X windows xterm window. If this option does not appear in `.ENTrc`, a standard Smalltalk editor is used instead.

- `ENTERPRISE = <path>`.
  This option must appear. `<path>` is the full path name of a directory that contains the Enterprise scripts and executable files for the other system components like the compiler.

C.3. Other Versions

As well as the version of Enterprise described in this thesis, two previous versions exist. The first version is that described in [Won92], [Cha92], and [SSW92]. It used a similar analogy to a business organization, but different asset types. Replication and re-classification were integral parts of the asset type instead of separate attributes. The compiler did not support as many features as the current version, and the executive was based on the ISIS [ISIS92] library. The user interface had not been implemented, so the asset types and relationships were specified using an ASCII graph file. A similar file is still used by the current interface to communicate with the compiler and executive, but the syntax is different.

A second version was developed and distributed as Enterprise v1.0. It was described in [LLM92] and [LSS93]. It included the current set of asset types and the current programming model using independent replication and re-classification. The graph file format was the same as the current one described in Appendix A. The interface had been implemented, but some features were missing or in a different form than the current version. The compiler supported more features like passing arrays. The executive was based on NMP [NMP88] instead of ISIS, and the animation component of the interface was not included.

The current version, described in this thesis, has improved the interface and added the animation component.

Existing programs from other versions can be converted by building the appropriate directory structure for the program, writing a new graph file if necessary, and copying the graph file and source files into the proper directories. The interface will then recognize that the program exists and allow editing it. The directory structure, file extensions, and file locations were described in Chapter 4. The graph file format was described in Appendix A.